Theses and Dissertations   1. Thesis and Dissertation Collection, all items

1995-09

Design and specification of an object-oriented data manipulation language

Stephens, Michael W.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/35201

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DESIGN AND SPECIFICATION
OF AN OBJECT-ORIENTED
DATA MANIPULATION LANGUAGE

by

Michael W. Stephens

September 1995

Thesis Advisor: David K. Hsiao
Co-Advisor: C. Thomas Wu

Approved for public release; distribution is unlimited.

DTIC QUALITY INSPECTED 1

19960401 024

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 1995	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE
Design and Specification of an Object-Oriented Data Manipulation Language

5. FUNDING NUMBERS

6. AUTHOR(S)
Michael W. Stephens

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This purpose of this thesis is to develop the design and specifications of an Object-Oriented Data Manipulation Language (O-ODML) for an Object-Oriented Data Model/Language (O-ODM) constructed to test and demonstrate the Multimodel and Multilingual Database System at the Naval Postgraduate School Laboratory for Database Systems Research, Monterey California.

New database applications, such as images and graphics databases, scientific databases, engineering design and manufacturing (CAD/CAM and CIM), require complex objects capable of storing images or large textual items and defining nonstandard application-specific operations. Traditional data models/languages were designed for record keeping, inventory control, product assemblies and inference making. In these traditional models, information about such complex objects is often scattered over many relations or records, leading to a loss of direct correspondence between a real-world object and its database representation. [Ref. 8]

This thesis developed an O-ODML to include such features as object creation and destruction, search and retrieve queries, attribute-set operations, input/output operations and covering relationships. For compilation, the thesis includes the detailed specifications of the grammar, production rules, syntax, and symbols for the O-ODML. Thus the O-ODML and O-ODM incorporate the ability to construct data structures that maintain the functional persistence of data, to specify intrinsic methods for manipulating data and to create objects with both ordinary attributes as well as sub-objects of sets that are independent of the original object.

14. SUBJECT TERMS
Object-Oriented Data Manipulation Language
Object-Oriented Data Model
Object-Oriented Database

15. NUMBER OF PAGES
105

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

DESIGN AND SPECIFICATION OF AN OBJECT-ORIENTED DATA MANIPULATION LANGUAGE

Michael W. Stephens
Lieutenant Commander, United States Navy
B.S. Southern Illinois University at Carbondale, 1982

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

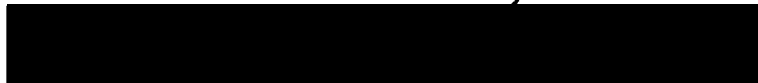## NAVAL POSTGRADUATE SCHOOL

September 1995
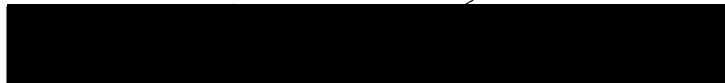
Author:

Michael W. Stephens

Approved by:

David K. Hsiao, Thesis Advisor

C. Thomas Wu, Co-Advisor

Ted Lewis, Chairman
Department of Computer Science

ABSTRACT

This purpose of this thesis is to develop the design and specifications of an Object-Oriented Data Manipulation Language (O-ODML) for an Object-Oriented Data Model/ Language (O-ODM) constructed to test and demonstrate the Multimodel and Multilingual Database System at the Naval Postgraduate School Laboratory for Database Systems Research, Monterey California.

New database applications, such as images and graphics databases, scientific databases, engineering design and manufacturing (CAD/CAM and CIM), require complex objects capable of storing images or large textual items and defining nonstandard application-specific operations. Traditional data models/languages were designed for record keeping, inventory control, product assemblies and inference making. In these traditional models, information about such complex objects is often scattered over many relations or records, leading to a loss of direct correspondence between a real-world object and its database representation. [Ref. 8]

This thesis developed an O-ODML to include such features as object creation and destruction, search and retrieve queries, attribute-set operations, input/output operations and covering relationships. For compilation, the thesis includes the detailed specifications of the grammar, production rules, syntax, and symbols for the O-ODML. Thus the O-ODML and O-ODM incorporate the ability to construct data structures that maintain the functional persistence of data, to specify intrinsic methods for manipulating data and to create objects with both ordinary attributes as well as sub-objects of sets that are independent of the original object.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ABDBMS	Attribute Based Database Management System
ASCII	American Standard Code for Information Interchange
BNF	Backus-Naur Form
CAD	Computer-Aided Design
CASE	Computer-Aided Software Engineering
CAM	Computer-Aided Manufacturing
CIM	Computer-Integrated Manufacture
CMAC	Cross-Model Accessing Capabilities
COBOL	Common Business Oriented Language
CODASYL	Committee on Data Systems Languages
DBMS	Database Management System
DBTG	Data Base Task Group
LR(k)	Left-Right(lookahead) Parsing Technique
M^2DBMS	Multimodel and Multilingual Database System
MDBMS	Multimodel and Multilingual Database System (M^2DBMS) with Cross-Model Accessing Capabilities (CMAC)
OID	Object Identifier
O-ODBMS	Object-Oriented Database Management System
O-ODDL	Object-Oriented Data Definition Language
OODL	Object-Oriented Data Language
O-ODM	Object-Oriented Data Model
O-ODML	Object-Oriented Data Manipulation Language
O-OPL	Object-Oriented Programming Language

ACKNOWLEDGEMENTS

I. INTRODUCTION

The purpose of this thesis is to develop the design and specifications of the Object-Oriented Data Manipulation Language (O-ODML). The purpose of the O-ODML is to provide a high-level query language for writing and processing transactions that can be executed in the object-oriented language/paradigm within the Multimodel and Multilingual Database System (M^2DBMS) via the Object-Oriented Model/Language (O-ODM) interface. The O-ODML, with the Object-Oriented Data Definition Language (O-ODDL), comprise the model and the language of the Object-Oriented Data Language (OODL) within the M^2DBMS. The M^2DBMS is the database research laboratory at Naval Postgraduate School Monterey, which provides a research platform for exploring solutions to the problems of heterogeneous databases and design concepts required for the development of a consolidated database system. The O-ODML is one of five model/language interfaces currently implemented in support of the M^2DBMS research laboratory.

A. MOTIVATION

The historical development of the traditional database systems proceeded through various data models as a variety of different types of applications evolved for the uses of databases and data relationships. Different types of applications led to the development of different data models and database systems to provide better support for the properties and data interrelationships required by these applications. These traditional database applications can be characterized as record keeping, product assembly and inventory control, as well as the more modern database systems developed for inference making.

The types of traditional heterogeneous database systems that have been developed and are in widespread use include the network, hierarchical, relational and functional data

1

models. Each of these types of database systems have been designed and developed with their own unique language, commands, structures and environments. These properties were designed to provide better support and to optimize the divergent types of data, storage and manipulation requirements. The network, commonly referred to as the CODASYL or DBTG network model, was developed to support inventory management, utilizing records and sets for its data structures and constructs. The hierarchical data model found its main use in the representation of the many types of hierarchical organizations, such as corporate structures, classification schemes and government organizations. The hierarchical model utilizes records and parent-child relationships to define its data structures. The relational data model is a popular model that has found many widespread applications in commercial industry. Its data structure is based on the relation and is usually defined using tables, tuples and attributes. The functional data model demonstrates the ability to use inference making by using mathematical functions as the modeling constructs. The functional data model utilizes entities and functional relationships for its standard data structure, with the function call returning the desired information. Each of these traditional data models are well suited for the traditional database applications for which they were first designed; however, newer applications have exceeded the capability of these traditional data models to structure complex abstract objects economically.

Newer applications and databases have been developed that require complex data structures. These data structures include those capable of storing images or large textual items and defining nonstandard application-specific operations. These types of databases include such applications as: images and graphics databases, scientific databases, engineering design and manufacturing applications (CAD/CAM and CIM), and software engineering and design (CASE). The O-ODM is well suited for these types of applications.

The O-ODM is the most recently developed data model. An object-oriented database system offers greater flexibility in giving the designers the capability to specify, both

2

the structure of complex abstract objects and the manipulations that can be applied to these objects. The O-ODM has incorporated many of the concepts of object-oriented programming language (O-OPL). These concepts include: the ability to create complex abstract class structures, to derive new classes through inheritance, to create unique objects and to encapsulate class methods in the objects with the data that they may modify. In addition, other O-OPL concepts adopted in the O-ODM include composition, code-reuse and polymorphism. [Ref. 8]

The growth in the number of models of heterogeneous database systems reflects industry efforts to achieve optimization in the ability to process and manipulate the unique relationships of the divergent types of data and their applications. While these efforts have achieved their intended goals, government and industry are now coping with the inability to cross-communicate between these database systems or to share data and resources between the different types of databases.

Considerable resources have been expended in developing the different types of data models and database systems. Each of these database systems has developed a following of dedicated user groups and supporters and each has become an integral part of our database resources. As important as these different database systems have become, none can be abandoned without suffering considerable economic losses and human effort. The data they contain and control has become regarded as invaluable resources with incalculable monetary value. However, their value is diminished by the lack of interoperability between the different database management systems. The languages and commands of one database system are incompatible with other data models. Users trained in the language of one database management system are generally incapable of accessing the databases constructed under a different system, without extensive cross training. The data stored in one system cannot be accessed or manipulated by users operating a different system. Nor can the database systems be readily converted to another system without great expenditure in

3

economic and human resources. These restrictions have long been recognized as limiting the potential uses for database systems overall. Several studies have been undertaken to resolve the issue in an effort to make database systems interoperable with other database systems. However, to make a database systems interoperable with another system, research suggests that either the data must be converted or a language interface must be built between each database management system and each database. While converting a database is possible, the costs of such conversions are expensive. Studies have also shown that making a system interoperable with another system through a language interface is also possible, but also expensive. Therefore, if such conversions and interfaces are to be cost effective, they must be limited to a minimum number of conversions or interfaces as possible.

To resolve this shortcoming, research at the Naval Postgraduate School Laboratory for Database Systems Research at Monterey California has led to the development of an experimental working model, the Multimodel and Multilingual Database System (M^2DBMS) with Cross-Model Accessing Capabilities (CMAC). The $M^2DBMS + CMAC$ has given users the capability to write transactions in one data language and retrieve or manipulate data in a database system that was originally constructed with a different database language. [Ref. 2]

The M^2DBMS with CMAC, or the MDBMS for short, was designed to solve two problems of heterogeneous databases and transactions.

- Data sharing or the ability to access data using a transaction written in one data language to access a database based on a different data model.
- Resource consolidation of all heterogeneous databases, system software and computer hardware.

The MDBMS was designed on the principle that only one initial conversion of the data to the format of the kernel database is required. That within the MDBMS system,

transactions conducted in any of the supported database languages are translated into the language of the kernel database system.

The O-ODM was developed and implemented to support the research being conducted with the MDBMS system. The MDBMS is in the final development phase required to demonstrate the capabilities of MDBMS using the five heterogeneous types of database systems.

B. THE THESIS ORGANIZATION

The Object-Oriented Data Model (O-ODM) consists of two parts, the Object-Oriented Data Definition Language (O-ODDL) and the Object-Oriented Data Manipulation Language (O-ODML). The O-ODDL has been developed and is described in a Master's Thesis by Bruce Badgett [Ref. 4]. The O-ODML design, specifications and constructs will be described in the following chapters.

This thesis is organized in two parts. The first part, consisting of Chapters II and III, contains design decisions and the specifications of the O-ODM and the O-ODML. The second part, consisting of Chapters IV and V, contains the formal specifications and constructs for object-oriented operations and queries.

The discussion on design decisions and specifications will explain the constructs for manipulating an object-oriented database system, describe the basic structure of such a system and considerations for manipulating data using object-oriented transactions within the framework of the O-ODM.

The formal specifications will describe the definitions, the grammar, the production rules and methods for composing complex or sequential manipulations, designed to make the language more robust.

5

The concluding remarks, related work and recommendations for future work are contained in Chapter VI. Appendixes A through D contain the specifications and production rules developed to support the design of the language and the compilation of transactions written in the O-ODML.

An object-oriented database was designed and constructed for the purpose of testing and demonstrating the concepts of the O-ODM. To facilitate the illustrations and provide explanations of design decisions and specifications for the O-ODML, the Logical Diagram (physical view) and Class Specifications for a Faculty-Student Database are included in Appendixes E and F. The Logical Diagram describes the relationships between the different classes and attributes. The Class Specifications define each class and provides a schema representation for the database.

Numerous queries were constructed for the Faculty-Student Database to assist in developing and testing the constructs of the manipulation language. To illustrate the methodology of various design and specification issues, various types of queries are included in Appendixes G and H.

II. OBJECT-ORIENTED CONSTRUCTS FOR MANIPULATION

The concept of object-oriented databases originated from the object-oriented programming languages in which data and behavior are strongly linked [Ref. 1]. Many of the concepts and structures associated with object-oriented programming languages can be adopted and further developed in an object-oriented data model (O-ODM) to provide the capability for more powerful data abstractions and structures required for newer, more complex applications. [Ref. 3] These O-OPL concepts and structures include:

- the ability to create abstract data structures
- the ability to create new and unique objects of a class type
- code reutilization through class inheritance and composition
- encapsulation of data and their behavior
- hidden implementation details

Where, in object-oriented programming, the basic concepts of the programming languages are founded on classes and functions, in the M^2DBMS, the O-ODM is constructed using classes, methods and operations [Ref. 1]. A key feature of object-oriented databases and the O-ODM is the power it provides the database designer to specify both the structure of complex classes and objects as well as the manipulations that can be applied to these objects. Another key feature is that class objects may have an internal data structure of an arbitrary complexity in order to contain all the significant information that describes the objects. Complexity is a relative term, used here it is intended to characterize the various data types that might be required in a data aggregate. These data types include the features normally found in a traditional database, but also offer additional and more powerful data abstractions and structures for newer, non-traditional applications [Ref. 1]. In traditional database systems, such information about a complex object is often dispersed

7

over many relations or records, resulting in a loss of direct correspondence between a real-world object and its database representation [Ref. 8].

A. CONSTRUCTS FOR CLASSES AND OBJECTS

In an object-oriented data model, the basic building blocks and concepts on which the database is modeled are classes, objects and abstraction. *Abstraction* gives the user the ability to create and build a whole idea, or an object, by defining its individual properties or parts. [Refs. 1, 6]

A *class* is an abstract data type or a data structure, defined by the database designer, to provide the specifications for the objects derived from the class. The class defines the class relationships and the class members. Class relationships may include inheritance, composition and covering. Class members may include methods and attributes.

A *method* is a type of data manipulation, contained within the object and defined by the class of which it is a member. Methods are only permitted to manipulate objects and attributes for which they are specifically defined. A method defines the means to interface with an object and provides a means of *encapsulating* or packaging within an object, both the data and the functionality of those allowable manipulations permitted by the object and/or its attributes. A method is one of two types of transactions (methods and operations) permitted by the O-ODM to manipulate data contained in the database. [Ref. 4]

Attributes are the specifications for the types of data an object may contain. An attribute is also a class member. It defined in two parts, the *attribute_name* (or *identifier*) and the attribute_type the attribute will represent. An *attribute_type* is a predefined data type and may be either a primitive data type or a complex data type. A *primitive data type* is a set of values of a type integer, float or character string. A *complex data type* is a set of

values of a type class. Object attributes are analogous of a record field in a hierarchical model, a tuple attribute in a relational model, or a record data item in a network model.

An *object* is derived from and defined by the class structure from which it is specified. A single object is a specific *instance* of a class, created to represent a new *entity* of data being added to the database. Every entity must be represented by a single object. The object then, is a collection of all the attributes and methods previously defined by the class from which the object is derived. [Refs. 1, 4]

An example of a class specification for the O-ODM is illustrated in Figure 1. A class is defined by its class name, any specified relationships, the attributes contained in the class and the methods defined by the class. Attributes are defined by specifying the attribute_name with the attribute_type. Methods are created by specifying the return_type (data type) with the method_name and syntax for manipulation. [Ref. 4] For more information on methods, refer to paragraph D.1. For information on relationships, refer to paragraph C. Figure 1 further illustrates the object-oriented paradigm where data and their behavior are closely linked by encapsulating the methods with the data types they modify within the specifications for the class.

Class *Class_name* : Specified Relationships {
    attribute_type$_1$ *attribute_name$_1$*;

    .

    attribute_type$_m$ *attribute_name$_m$*;
    return_type$_1$ *method_name$_1$*;

    .

    return_type$_n$ *method_name$_n$*;
};

Figure 1. Specification of a Class

B. OBJECT IDENTIFIERS

The *Object Identifier* (OID) is a hidden attribute and provided to identify each individual object stored in the database with an identifier or primary key. The value of the OID is not visible to the end-user and may only be referenced indirectly. Each OID is generated by the system and assigned to an object by the DBMS when the object is first created. The OID may then be used as a reference to identify that specific object [Refs. 1, 4]. It is unique in that no other object in the database may have the same OID. The OID is *immutable*; that is, it is permanently assigned to the newly created object and will not change for the life of the object. Nor will the OID be reassigned to another object after the original object is deleted from the database.

C. OBJECT-ORIENTED CLASS RELATIONSHIPS

1. Class Composition

A *complex class* is a class that contains a different class as an attribute. In the O-OPL, this class relationship is referred to as *composition* and is adopted for use in the design of the O-ODM. Composition provides the user the ability to create a class with attributes and methods, and then include that class as an attribute in yet another class. This is referred to as a *"has a"* relationship, where a class has a class as an attribute. The specifications for a complex class are shown in Figure 2 where the class included within another class is illustrated as *Class_name$_2$*. This relationship will be used in the Faculty-Student Database, illustrated in Appendix E and F, and its usage (attribute-object) will be discussed further in Chapter III.

```
Class Class_name_1{
    attribute_type_1        attribute_name_1;
    :
    attribute_type_m        attribute_name_m;
    Class_name_2            attribute_name;
    return_type_n           method_name_n;
};
```

Figure 2. Specification of a Complex Class

2. Class Inheritance

Inheritance is a relationship between two or more classes where the *child class* will assume all the attributes and methods contained in the *parent class* in addition to any new attributes and methods defined for the child class. Inheritance provides the mechanism to derive a new class from an existing class without rewriting the code for those attributes and methods that are derived from the parent class.

A parent class may also be referred to as a *superclass* and is a *generalization* of zero, one or more child classes. The child class may be referred to as a *subclass* of the superclass and is a *specialization* of the generalization. Through inheritance, existing code specifying the attributes and methods of the parent class is reused in the child class where it may be either modified or added to, specifically tailored to the needs of a subclass. Thus, inheritance and code reuse provides for the O-ODM, the ability to design one parent class that can be reused by many subclasses of the parent class. To illustrate this point, refer to Figure 3.

In Figure 3, an object of either child Class *Student* or *Faculty* will inherit all the properties of the parent Class *Person*. Thus, an object of either Student or Faculty will inherit all the attributes and methods of Person, such as the *pname, paddress* and *sex* attributes. In addition to the inherited methods and attributes, the child classes may also

11

```
┌─────────────────────────────────────────────────────────┐
│                  ┌───────────────────────┐               │
│                  │ Class Person{         │               │
│                  │    Name     pname;    │               │
│                  │    Address  paddress; │               │
│                  │    char*    sex;      │               │
│                  │ };                    │               │
│                  │                       │               │
│                  └───────────────────────┘               │
│                    ↙               ↘                     │
│  ┌──────────────────────────┐  ┌──────────────────────────┐│
│  │ Class Student : inherit Person{│ Class Faculty : inherit Person{│
│  │    char*    student_no;  │  │    char*    dept;        ││
│  │    char*    major;       │  │    set_of  Course   teaches;││
│  │    set_of  Course  schedule;│ };                       ││
│  │ };                       │  │                          ││
│  └──────────────────────────┘  └──────────────────────────┘│
└─────────────────────────────────────────────────────────┘
```
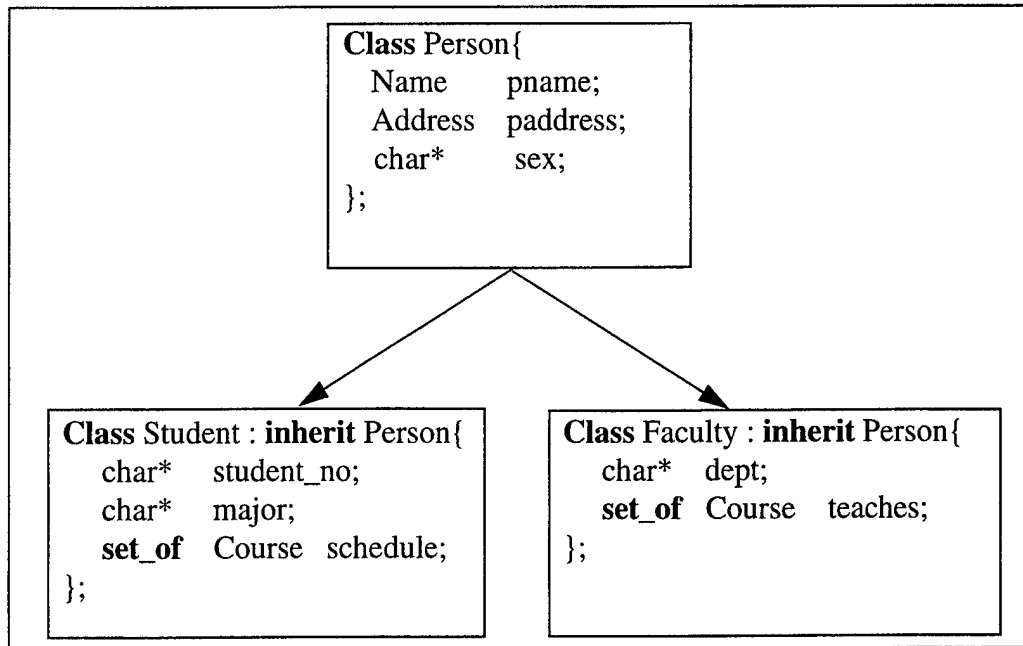
Figure 3. Inheritance Relationship

define additional new attributes and methods that are specific to the child class only. This is illustrated in both, the child Class Student where new attributes are created for *student_no*, *major* and *schedule*, and in the child Class Faculty where new attributes are created for *dept* and *teaches*. Intuitively we see that a student is a person with all the properties and attributes of a person. A member of the faculty is also a person with all the properties and attributes of a person. For this reason, inheritance is commonly referred to as an *"is a"* relationship. As expected, in addition to being a person, students and faculties have additional attributes that distinguish them as either a student or a faculty.

3. Covering

A Covering relationship is said to exist when every object from one class (A), corresponds to one or more objects of a second class (B). A being said to cover B. The covering relationship does not have to partition the second class, i.e., two objects from the first class (A) may correspond to one or more of the same objects in the second class (B). Thus, cov-

ering provides a "one-to-many" (1:M) relationship between an object of one class (A) to many objects of a second class (B) (*from-object-to-class*) as shown in Figure 4.
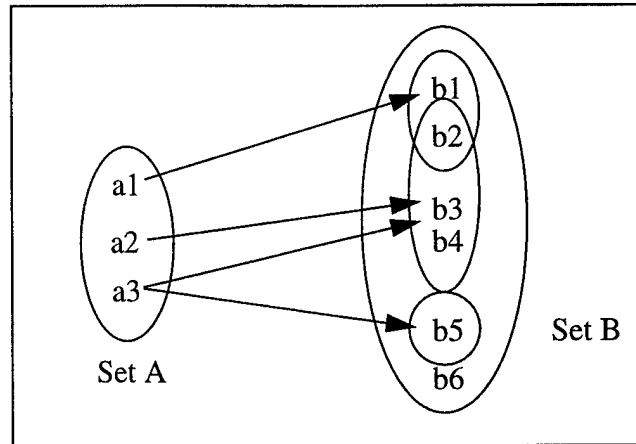
Figure 4. Covering Relationship (1:M)

Now consider the individual objects of the second class (B). Each object of class (B) possesses all the characteristics and properties of the class and is said to generalize class (B). Considered together as a class, all the objects of B form a singleton with all the characteristics and properties of the class. Now, every object from the first class (A) corresponds to the second class (B). With this type of correspondence, there are many objects of A covering the class B (*from-class-to-class*) as shown in Figure 5, thus providing a "many-to-one" (M:1) relationship.
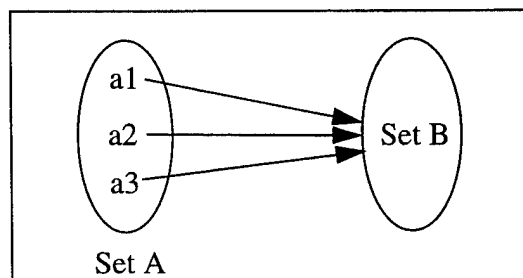
Figure 5. Covering Relationship (M:1)

In the Faculty-Student Database included in Appendixes E and F, the cover relationship is implemented between Class Team and Class Student. This is illustrated in the logical diagram shown in Figure 6, in which each team of the cover class, maps to one or more students of the member class. "Team-stu" is shown to represent the relationship between the objects of the two classes.
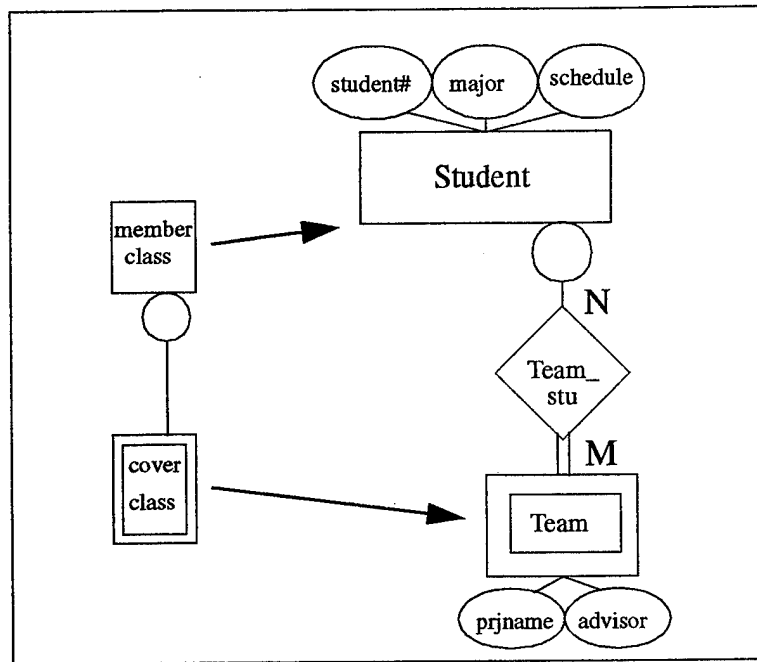
Figure 6. Covering Relationship Logical Diagram

Figure 7 illustrates the class specifications for a cover class in which Class Team covers the member class, Class Student. In Class Team, the cover relationship is specified by a colon, the reserved word COVER, and the specified member's Class_name.

In the underlying data structure, a separate internal data table is created to store each relationship between an object of the cover class Team and objects of the member class Student. This internal relationship is represented in the Faculty-Student Logical Diagram in Figure 6 as "Team-stu". The internal data table is depicted in Figure 7 as the Team-stu Data Table. For each such relationship, a new object is created in the internal table recording the

object OID of the new relationship, the OID of the team object and the OID of the student object assigned to the team. As new students are assigned to a team, new objects are created for the relationship. Note that this underlying data structure is not a part of the conceptual view and not visible to the user. However, in the constructs of the manipulation language, provisions are required and implemented to manipulate the data of the cover relationship utilizing this internal table. These operations will be discussed in Chapter III.E.

Class Student : **Inherit** Person{
    char*   student_no;
    char*   major;
    set_of Course   schedule;
};

Class Team : **Cover** Student{
    char*   prjname;
    set_of Civ_Fac   advisor;
};

Team-stu Data Table		
OID(_relationship)	OID_s(tudent)	OID_t(eam)
SOS1	P1	T1
SOS2	P2	T1
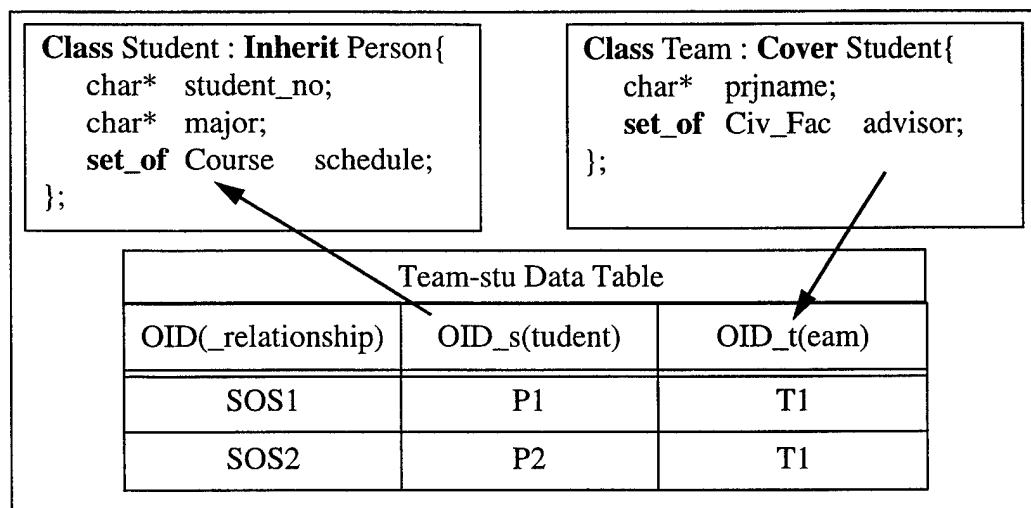
Figure 7. Covering Implementation

D. OBJECT-ORIENTED MANIPULATIONS

1. Methods

A method is a transaction, defined for and encapsulated with the data it will modify, within a class data structure at the time the class is specified. Methods are created by specifying its return_type (data type) with its method_name and syntax, as illustrated previously in Figure 1. With a method, we bring operations or functions into a database and associate them with specific objects and attributes of the database. The preferred means of manipulating an object or its attributes is by using an encapsulated method; defined to ensure that

only legitimate operations or functions may be performed on objects of class. Methods also provide a standard interface for the database designer to form a complex object of interfaced objects. Restricting access to the object and attributes, through a standard interface specified by the programmer, the operational and data integrity of an object can then be more assured. Thus, a method will only perform transactions on attributes contained within an object as originally specified by the object class. A method is invoked by sending a message to the object to execute the corresponding method.

In the Object-Oriented Data Model, the design and specifications for methods as well as attributes are further defined in the Object-Oriented Data Definition Language. [Ref. 4]

2. Operations

Object-oriented *operations* are transactions, defined for use on many objects and different classes, but are not methods contained within any object class of the database. These operations are managed separately from the database by the O-ODBMS. They are not transactions written in terms of specific operations or functions predefined for individual data aggregates. They are general operations designed to manipulate all the objects and their attributes stored in the object-oriented database.

Operations are reminiscent of functions written in an object-oriented programming language and give the user additional freedom by adding functionality to the DBMS after the data aggregates have been defined. In the design of the object-oriented data manipulation language, operations are specified for the following types of transactions:

- object creation and deletion
- search and retrieval
- input and output
- attribute value assignment

16

- statistical operations

- set operations.

Table 1 lists the set of O-ODML operations, their syntax and semantics. These will be discussed further in Chapters III and IV.

    Additional programming language features were also implemented to facilitate conducting query transactions and to facilitate the user interface with the database management system (DBMS). These include decision statements, iteration loops, and object member access rules. These features will be discussed in Chapter IV.

Operations	Syntax	Semantics
insert	insert *class_name*	Create a new object of a class and add it to the set of objects within a class and the database table.
delete	delete (*obj_ref*)	Delete *variable_name* object from its class and database table. [Delete implies destruction]
	delete (*set_attr, single_value*)	Delete a single *attribute* from a *set_attr*.
find_One	find_One *class_name* where expression	Searches and returns only the first object from *class_name* that satisfies the expression.
find_Many	find_Many *class_name* where expression	Same as find_One, but will return all objects, as a set, that satisfy the expression.
	find_Many *class_name*	Same as find_Many, but returns all objects in the set.
read_Input	read_Input (*var_list*)	Read the input from the user, the values to be assigned to the *var_list*. Primitive data only.
display	display (*obj_ref.attribute*)	Display the value of an *attribute* of an object on screen when the *attribute* is a primitive data.
	display (*obj_ref*)	Displays all the primitive *attributes* of an object. (does not included inherited *attributes*)
	display ("*string_literal*")	Displays text strings.
add	add (*set_attr, single_value*)	Add a *single_value* to a list or set of *attributes*.
count	count (*set_attr*) or count (*class_name*)	Returns the number of elements in the given set.
project	project *attribute*	Only allows projection of single *attribute*.
min	min (*set_attr*)	Returns the min numerical value in a set.
max	max (*set_attr*)	Returns the max numerical value in a set.
avg	avg (*set_attr*)	Returns the avg numerical value in a set.
contains	*set_attr* contains expression or *set_attr* contains *obj_ref*	Set comparison function. Returns objects in a set satisfying an expression or matching objects in second set.

Table 1. List of Object-Oriented Operations

18

III. OBJECT-ORIENTED OPERATIONS

The structure of an object can be said to be divided into visible and hidden attributes. Visible attributes may be directly accessed for reading by operations external to the object, or by a high-level query language. The hidden attributes of an object are completely encapsulated, and can only be accessed through predefined encapsulated methods.

While some object-oriented data models require that all legitimate transactions must be predefined as a method within an object, this forces a *total encapsulation* of the objects and data model. Total encapsulation would restrict the available transactions to only those predefined as methods within a class. This may be well suited for some purposes; however, it is too restrictive for most database applications. An object-oriented database that contains operations is more robust in that it allows greater flexibility in writing common transactions that are available for use on all classes, objects and attributes.

Operations are database transactions, defined for use on one or more class of objects by the application programmer. These operations can be defined at the time the database is first created or they may be added later. Compared with a method, operations are of a more general nature and allow the database to be manipulated by some combination of the four primary types of database transactions: Insert, Delete, Retrieve and Update. These types of operations are managed by the DBMS. They are not a method, predefined within the data structure of any specific object, but are a part of the entire database and applicable for all objects of the database.

Transactions may be formed by either a single operation or method, or they may contain many operations and methods. Whether a transaction contains a combination of a single or many operations and methods, transactions must be composed in the form of a query.

A *query* is a structured collection of declarations and operations in a block format. Each query must be compiled by the O-ODML compiler prior to processing the requested transactions. For the compiler to process the query, each query must be formatted and structured within strict guidelines in order to be recognized by the compiler as a legitimate query. These guidelines will be discussed in Chapter V, The Query Constructs.

A. OBJECT CREATION AND DESTRUCTION

The Insert Operation is used to create a new object of an object class. Since a class defines a type, a schema specification and a collection of objects of the class type, inserting an object means adding a new object of the class type to the collection of objects in the class. Deletion of the object from the collection of objects in the class implies destruction and removal of the object from the database.

1. The Insert Operation

An object is created by the *Insert Operation* specifying the operation and the name of the object class. A new object is then created in the database with all the predefined attributes and methods of the object class. When the object is first created, with the exception of the system generated Object Identifier (OID), no values are assigned to the attributes of the object. All other attribute values are assigned by the user with specific attribute assignment operations after the object is created and stored in the database. The syntax and format for the Insert Operation are as follows:

<p align="center">*variable_name* := insert *class_name*;</p>

In the example above, the newly created object will be of a class type specified by class_name (introduced in Chapter II). The *variable_name* type must be predefined as an *object reference* and is restricted to representing a singular OID. In the O-ODM an object reference is a data_type, denoted as *obj_ref*, and may only represent a single OID value.

This is appropriate for an Insert Operation as the operation may only create one object at a time. As the system creates the new object, a new OID will be generated and assigned to the object. To permit further operations with the new object, the value of the new OID must be assigned to the identifier, variable_name, with an assignment operation at the time the object is first created.

The *Assignment Operator*, ":=" is required during the initial Insert Operation to maintain control of the newly created object. With no other attributes assigned to the new object and the OID invisible to the user, there is no economical way to retrieve the new object. Thus, if an object could be created with just the syntax;

insert *class_name*;

then, to retrieve the new object, all the objects of that class must be retrieved and the newly created object selected from the entire set of objects. Even then, with no other attributes assigned, the user would have difficulty in locating and selecting the newly created object for further manipulations. To prevent this from occurring in the O-ODM, the production rules are designed to ensure the correct format and syntax are observed.

2. The Delete Operation

Delete implies destruction in the object-oriented syntax. The *Delete Operation* will remove the object from the class of objects contained within a class type. Once removed from the class, it can no longer be manipulated by the user by any method or operation. With the object deleted from the database, all attributes assigned to the object are also removed from the database with the exception of the *set_of* attribute discussed in the following paragraphs. Even in the case of a complex object, where an attribute of an object is also another object of a different class, that *attribute-object* is removed from the database as well. This situation would arise where the user desires to remove a person from the Class Person. Class Name and Class Address are attributes of Class Person. If a person is

removed from the database, the person's name and address attribute-objects will also be removed from the database.

By design, data in the O-ODM database is persistent. To ensure the integrity of the database, a Delete Operation can only be evoked explicitly. The syntax and format for the Delete Operation are illustrated as follows:

delete (*obj_ref*);

In this example, obj_ref would be represented by some variable_name whose type must be predefined as an object reference, identifying a single OID. The Delete Operation is a singular operation. It can only delete a single object from a class of objects. The exception was previously noted where the deletion of a complex object will also delete any assigned attribute-objects of a different class as well.

The second format for the Delete Operation is designed for Set Operations. This is the case where an attribute of an object is defined as an *attribute-set*, either of an attribute_type *set_of* or *inverse_of*. Both set_of and inverse_of represent a set of objects containing zero, one or more attributes in the attribute_set. Each attribute listed in an attribute-set is an object. As an object, they are created with an Insert Operation within their own Class. These objects may later be assigned as an attribute to an attribute-set of one object or assigned to several objects and their attribute-sets. Once a value is added to the attribute-set, the Delete Operation can be used to remove a single attribute value (object) from the set of attributes contained in the attribute-set. In this case, delete does not imply destruction. This is an example of polymorphism, or operator overloading adopted from the O-OPL. Here, the operation's name "Delete" is overloaded and the specific usage is called, based on the operation's *signature*. The signature contains the name of the operation and the list of arguments in the operation's parameter list. An example of the usage of the Delete Operation removing an attribute from an attribute-set is as follows:

delete (*set_attr, single_value*);

Where *set_attr* represents an attribute-set of a specified object and *single_value* is an identifier, representing the OID of one object; then, the single object would be removed from the list of objects contained in the single object's attribute-set. In this mode, Delete does not destroy the object removed from the set as the removed object may be assigned as an attribute to an attribute-set in many different objects. The intent is simply to remove the object from a set of an attribute-set of a single object. As an example, in the Faculty-Student Database, a Student object may be assigned to the rosters of several different Course objects. One attribute for the Class Courses is the set of students assigned to that course. A student in the database may be assigned to several different courses. Each course would have that student listed as one of the students in its attribute-set. In the case where the student is to be removed from a single course, then, the transaction,

delete (*course_object.roster, single_student_OID_value*);

will only remove a single student from the list of students enrolled in that course. However, the student object representing the student is not deleted from the database, and it is possible for the same student to still be assigned to several other courses. For more information on Set Operations, see paragraph F.

B. VALUE INPUT AND OUTPUT

Attributes and data values cannot be manipulated unless they are first stored in main memory. There are three means of entering a data value into memory.

- assign the value to a variable as it is read from a query file
- retrieve the value from storage and assign it to a variable, a *retrieve operation*
- read the value entered by the user, *an input operation*

23

In order to view the results of a transaction, the values of selected attributes or objects can be written, either to the terminal screen or to a file, an *output operation.*

Transactions, previously written in the format of a query and saved as a file is the most common method for performing operations on the database. File queries may be pre-tested, used, modified and reused many times. Values may be assigned to a variable during the course of the query and then further manipulated as part of the query's database trans-actions. More commonly, a query may retrieve a value from storage and assign the retrieved value to a variable for further manipulations. If desired, a query can be written in such a way that different variations of the query can be formed by changing the values of the query arguments for each execution of the query. Another method of entering values is the Read_Input Operation, where a user enters values during the execution of the query.

1. The Read_Input Operation

The *Read_Input Operation* is an interactive method of reading data values, pro-vided by the user from the terminal keyboard, into a query as it is executing. The Read_Input Operation will temporarily cause the execution of the query to pause while waiting for the user to provide a correct value or argument for the variable or variables listed in the parameters for var_list. The *var_list* is a list of one or more identifiers repre-senting one or more variables. Once prompted, the user must enter a valid value prior to the query proceeding with the query execution. A common practice is for the query program-mer to use a Display Operation to prompt or advise the user when or what type of valid arguments to enter. Without such a prompt, the query execution would pause awaiting an input from the user, but the user would be unaware a response was required or why the pro-gram execution appeared to halt. The syntax and format for the Read_Input Operation are as follows:

      read_Input (var_list);

Each variable listed in var_list will be assigned a value in sequential order as it is entered from the terminal. As an illustration, suppose the user is entering a name into the database and the following operation is encountered in the execution of the query:

read_Input (a, b, c);

After the program reads and executes the Read_Input Operation, the query will pause in execution and wait for an input from the user, three inputs are required in this case, one for each of the three variables (a, b and c). The user will first enter a value of a type defined in the query declarations for the variable "a". For an attribute of a last name that is of type char*, the user would enter the following at the terminal keyboard:

Stephens<carriage return>

At the carriage return, the Read_Input Operation will read and store the character string "Stephens" in the first variable (a) listed in the var_list and await the next entry. This will continue until each variable listed in the var_list, has been assigned a value, as read from the keyboard by the Read_Input Operation.

After the last value is read, the Assignment Statement must then be used to assign each text string value to a proper attribute. The next operation will then be an Assignment Statement, to assign a value entered at the terminal keyboard for the variable "a" to the correct attribute. If the three variables of the var_list (a, b, c) are of type, "char*" and the user had identified a Person object to which he wanted to enter the last name, first name and middle initial, the next three operations of the query would read:

```
person_variable.name.lname := a;
person_variable.name.fname := b;
person_variable.name.mi := c;
```

2. The Display Operation

The *Display Operation* is an output operation. It displays to the terminal monitor and to the user, information, data and messages that the query programmer wishes to convey. It does not interact with the data stored in the database in any way, other than to display data that may have been retrieved or processed by other methods or operations.

The programmer, building a series of complex methods and operations in a single query, may choose to display the results of each database transaction, or to display the final results of a series of transactions. The programmer may also use the Display Operation to prompt the user for keyboard data entries, as in a Read_Input Operation, or to display advisory messages and exceptions to the user via the monitor. Note, when programming a text string literal for display, the text string is enclosed in double quotation marks, i.e., *"text-string"*. The Display Operation may be used in one of three ways:

- Display the primitive value of one or more attributes of an object
- Display the primitive value of all the attributes of an object
- Display a text string to the monitor

There are several limitations on the capability of the Display Operation. To display an attribute of an OID, the OID value must first be assigned to a variable where it is used by the Display Operation to specifically identify which object's attributes are to be displayed. The Display Operation will only display *primitive* data. Primitive data may only represent a single value. If asked to display all the attributes of an object, the Display Operation will omit displaying the attributes listed in an attribute-set or the attributes of an attribute-object. These are separate objects with their own set of attributes. For example, if asked to display all the attributes of an object of Class Person, it will only display the primitive data contained in the attribute "sex". The attributes, pname and paddress, are class objects, Name and Address, containing their own attributes. Thus, the Display Operation is

26

limited in that it will not automatically follow a reference to another object and its attributes. These attributes can be displayed, but they must be specified using the *member access operator* (*dot notation*) to be discussed in Chapter IV.C. This limitation will also apply to those attributes inherited from a parent class. The syntax and format for the Display Operations are as illustrated below. Note how the member access rule is applied in the following examples. The format for the member access operator is a period between the object reference and its attribute.

- To display the value of one or more attributes to the screen where the attributes contain only primitive data

    display (obj_ref.attribute$_1$);
        or
    display (obj_ref.attribute$_1$, obj_ref. attribute$_2$);

- To display all the primitive attributes of an object

    display (obj_ref);

- To display a text string or message to the screen (quotation marks are required around the text string literal)

    display ("string_literal");

- To display an attribute of a class attribute contained in an object where attribute_object represents an OID of an object that is an attribute

    display (obj_ref.*attribute_object*.attribute$_1$);

- To display an inherited attribute of a parent class where class_name represents an OID of an object with attributes that are inherited

    display (obj_ref.*class_name*.attribute$_1$);

3. The Project Operation

The *Project Operation* is a special output operation designed for debugging purposes. When a retrieve operation identifies a specific object, Project permits the display or assignment of the attributes of an object reference or a single attribute of the object.

project(variable_name);

C. OBJECT RETRIEVAL

Object retrieval is often the first step of a query, designed to locate an object so that further manipulations may be performed on the object or its attributes. The retrieval is a three-step process:

- a systematic search of the database
- litigation of an object based on a *search condition*
- assign the object reference value to a variable for further manipulation

The search condition specifies a *Boolean expression* that the desired object or objects, and/or their attributes must satisfy. A Boolean expression is an expression that can only be evaluated as either true or false. The syntax for a Boolean expression requires the use of a *Relational Operator* to determine equality between two values. If the comparison is evaluated to be true, then the Boolean expression will return a value of true. If the Boolean expression is evaluated as false, then it will return a value of false. The Relational Operators include and are represented by the following symbols:

<	less than
>	greater than
=	equal to
/=	not equal to
>=	greater than or equal to
<=	less than or equal to

28

Where the search condition is satisfied, then the object's OID that satisfies the condition is identified and retrieved. In a more complex search, more than one search condition may be specified. The criteria for the multiple search conditions are satisfied, if the search also satisfies the *logical operators* "and" or "or", with each search condition.

There will be some searches where more than one object may satisfy the search conditions. The retrieve operations will either return the first object (OID) located, or will return all the objects (OIDs) located, that satisfy the parameters of the operation specified as well as the search criteria. The two retrieve operations are *Find_One* and *Find_Many*.

1. The Find_One Operation

The *Find_One Operation* is a search and retrieve operation that will only return the first object's OID that satisfies the search condition. There are no provisions to continue the search if the object returned is not the desired object. If a different object was desired, a different search condition must be specified and a new search initiated. Only one object may be manipulated at a time. Once located, the OID of the object is assigned to a variable of type obj_ref and stored in main memory for further operations. Obj_ref is of a type that will only store a single OID. If desired, more than one object can be retrieved in separate searches. Each OID must be stored in a separate variable of type obj_ref. If this is the desire, a different search condition must be specified to preclude retrieving the same object first found, during subsequent search operations.

The Find_One Operation mandates that a search condition must be included for the operation. A search condition is always specified by using the reserved word WHERE. The syntax and format for the Find_One Operations are as follows:

obj_ref := find_One *class_name* where *search condition*;
    or
obj_ref := find_One *class_name*
        where *search condition$_1$* logical operator *search condition$_2$*;

2. The Find_Many Operation

The *Find_Many Operation* is also a search and retrieve operation. However, Find_Many will retrieve and store all object OIDs that satisfy the search conditions. Each OID may be stored in single variable of type obj_set. *Obj_set* is an attribute_type that is capable of storing a set of one or more OIDs.

The search condition for a Find_Many Operation is optional. Used without such a condition, Find_Many will return all the object OIDs of the class_name specified. Used with a search condition, the Find_Many Operation will use the condition to limit the number of OIDs returned. The Find_Many Operation will continue to search the database until all OIDs of objects that match the search condition are located and retrieved. The search condition in a Find_Many Operation is also specified by using the reserved word WHERE. As in the Find_One Operation, more than one search condition may be used and the criteria for the search conditions satisfied if the objects also satisfy the logical operators of "and" or "or", with each search condition. The syntax and format for the Find_Many Operations are as follows:

> obj_ref := find_many *class_name*;
>   or
> obj_ref := find_many *class_name* where *search condition*;
>   or
> obj_ref := find_many *class_name*
>   where *search condition₁* logical operator *search condition₂*;

D. OBJECT ATTRIBUTE ASSIGNMENT

Assignment of a value to an attribute of an object occurs as part of an *assignment expression* and two special operators, the *Assignment Operator* and the member access operator. The Assignment Operator is denoted by the combination of two symbols, the colon and the equal sign, ":=". An example of an Assignment Operation and an expression are as follows:

variable_name.pname.fname := 'Mike';
                    where
left-side attribute_type = right-side attribute type

In this example, the right-side of the assignment is evaluated. If it is of the same attribute_type as the attribute on the left-side of the assignment, then the value of the right-side is assigned to the left-side attribute. Note the use of the single quotation for the data type, char* value. Also, note the use of *variable_name* to represent an OID of an object. In the O-ODM, functional references to members of an object (attributes and methods) use the member access rule; however, a variable (of type obj_ref) whose value is the object's OID must be provided for access. In the previous example, a specific object of a class was retrieved and the OID assigned to *variable_name* of type obj_ref. This type of reference can be viewed as a shorthand reference to the object and temporarily stored in memory by the program for the life of the query. During the remainder of the query execution, further references to *variable_name* will always denote the specific object of the class previously located and assigned to *variable_name*. Attributes or methods of that object can then be referenced as *variable_name.attribute*. Assignments of values to that object's attribute would then be as follows:

variable_name.attribute := attribute_value;

In our first example,

variable_name.pname.fname := 'Mike';

variable_name represents the OID of the object we wish to a assign value. The dot notation denoted by the period between *variable_name* and pname accesses the class_attribute Name designated as pname in our Faculty-Student Database. One attribute of the attribute-object pname is fname, representing the first name of a person. In this example, the dot notation accesses fname, an attribute of pname.

E. COVERING OPERATIONS

Covering Operations are those operations where special provisions are required to conduct search operations involving those classes constructed with the covering principal. From Chapter II, Cover was defined to mean that an object of one class maps to a subset of objects of another class. While the search operations are not necessarily unique, a method is required to signify to the compiler that the search operation involves a cover class and requires a modified execution procedure. This special method is invoked using the reserved word, IT. IT is a special symbol that effectively becomes a self referencing pointer [Ref. 9], To illustrate the proper usage, refer to the following example taken from the Faculty-Student Database and shown in Figure 8:

$$variable\_name_1 := find\_many \text{ Team}$$
$$\text{where name} = \text{'DB' and } IT \text{ contains}$$
$$variable\_name_2\_of\_member\_class;$$

In the underlying data structure, this relationship is maintained in an internal table created to record each relationship between an object of the cover class and multiple objects of the member class. As previously explained in Chapter II.3.C., this relationship is maintained in the logical data structure of the database and is not visible to the user.

As shown in Figure 8, the query would first retrieve all OIDs of the students specified in the first search operation. The second search operation, the cover operation, would retrieve all the OIDs for Teams where those students found in the first search are team students. This relationship is found in the Team_stu internal data table.

```
Class Student : inherit Person{          Class Team : Cover Student{
    char*   student_no;                      char*   prjname;
    char*   major;                           set_of  Civ_Fac   advisor;
    set_of Course    schedule;           };
};
```

Team-stu Data Table		
OID(_relationship)	OID_s(tudent)	OID_t(eam)
SOS1	P1	T1
SOS2	P2	T1

```
a := find_many Student where
                pname.fname = 'Luis'
                or pname.fname = 'Recep';
b := find_many Team where IT
                contains a;
```
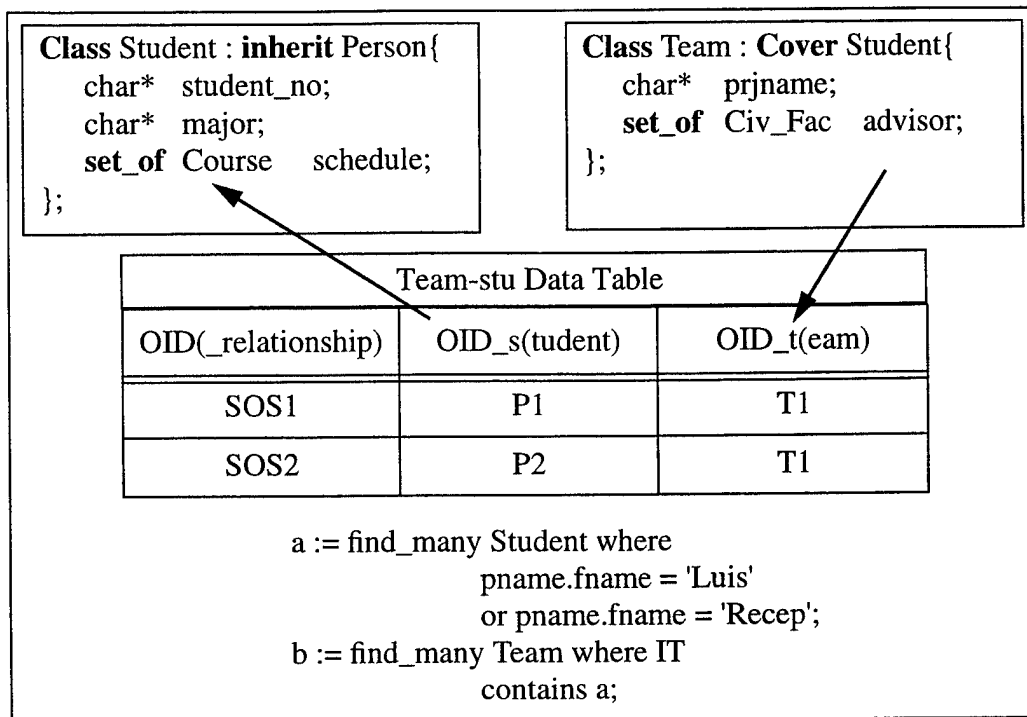
Figure 8. Covering Operation

F. SET OPERATIONS

Sets are special attributes of an object that may contain zero, one or more values (objects). These have previously been denoted as *attribute-sets*. To illustrate this point, refer to the O-ODM data model, the Faculty-Student Database in Appendixes E and F. One such illustration is Class Student. A student in the data model may be scheduled for either zero, one or many courses of the Class Course. Another such illustration is where a member of the Civilian_Faculty class may advise zero, one or many teams of the Class Team. The O-ODDL specifies this set relationship as a new data type denoted as *set_of* or *inverse_of* [Ref. 4].

To the user, *set_of* and *inverse_of* represent identical attribute-set concepts. Both represent sets of attributes and both attribute_types are created automatically as each new

object of a class containing an attribute-set is created. Attributes may then be added to and deleted from the set as necessary.

In the underlying data structure, a single attribute contained in an attribute-set (set_of or inverse_of) is a separate object of a different class, identified by its own OID, with its own attributes and methods. Inverse_of was implemented as the complement of set_of. This was accomplished by creating one internal data table where the relationships between objects of set_of and inverse_of are maintained. This implementation feature serves two purposes;

- to reduce the redundancy of storing the same information twice
- to preserves the integrity of the database by ensuring that any transaction on one object of an attribute-set will also be performed on its compliment.

This logical view of the database and underlying data structure is not visible to the user nor within the scope of this thesis; however, the explanation is necessary to help illustrate the following Set Operations.

1. The Add Operation

The *Add Operation* enters a single new attribute value, a previously created object, to the set of attributes contained in an object's attribute-set (set_of or inverse_of).

The syntax and format for the Add Operation are as follows:

add(*set_attr, single_value*);

In this example, *set_attr* identifies the specific object OID <u>and</u> its attribute-set that an object will be added to. Added to the set_attr, is an OID, represented by a *single_value* identifier of type obj_ref (attribute_type).

To illustrate, refer to Figures 9, The Attribute-Set Logical Diagram and Figure 10, Set-of/Inverse-of Attribute-Set Relationships. This is an excerpt from the Faculty-Student Database Logical Diagram and shows the relationship between Student-schedule and

34

Course-roster. Course-stu is the internal data table created to record and maintain this relationship.
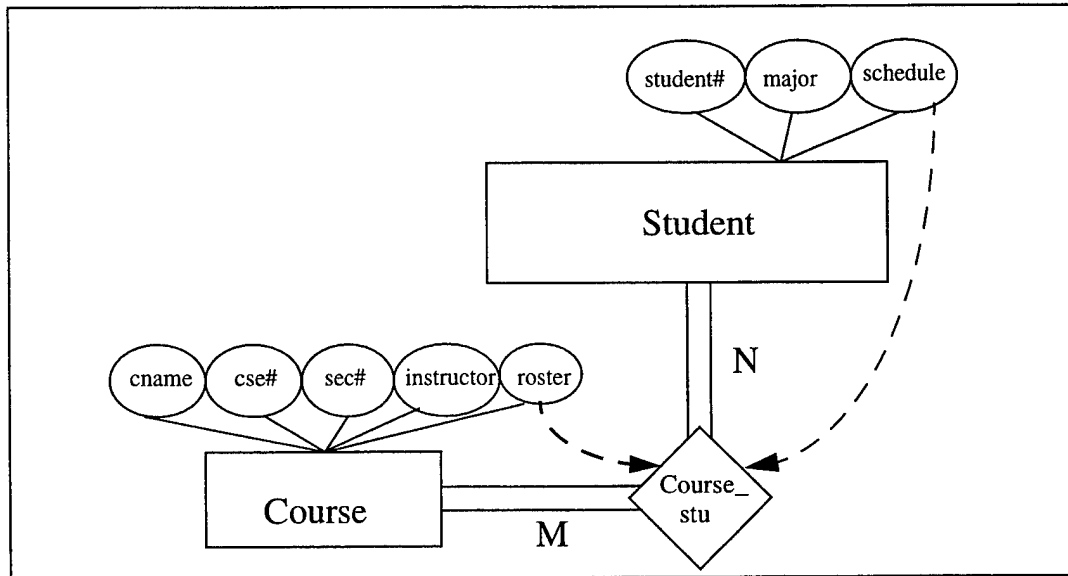
Figure 9. Attribute-Set Logical Diagram

Figure 10, illustrates the class specification for each class and the Course-stu internal table implemented to maintain the two attribute-set relationships.

The Add Operation can be used to add a student to a course-roster or to add a course to a student-schedule. In the underlying data structure, the internal table for the Course_Stu relationship will add a new object with its own OID for this single, one to one (1:1) relationship. The attributes for the object will consist of the OID for the student and the OID for the course, thus recording the relationship between the two attribute-sets. As new students are added to course-rosters, the roster/schedule relationship will automatically record the course in the student-schedule attribute-set as well. Many students may be entered on a course-roster, many courses may be listed in a student-schedule, thus establishing the "many-to-many" (M:M) relationship. With this relationship recorded in one location, any

```
Class Course{                          Class Student : inherit Person{
    char*    cname;                        char*    student_no;
    char*    cse_num                       char*    major;
    char*    sec_no;                       set_of Course    schedule;
    Faculty  instructor;               };
    inverse_of Student.schedule    roster;
};
```

Course_stu Data Table		
OID(relationship)	OID_c(ourse)	OID_s(tudent)
SOCS1	C1	P1
S0CS2	C2	P1
SOCS3	C4	P1
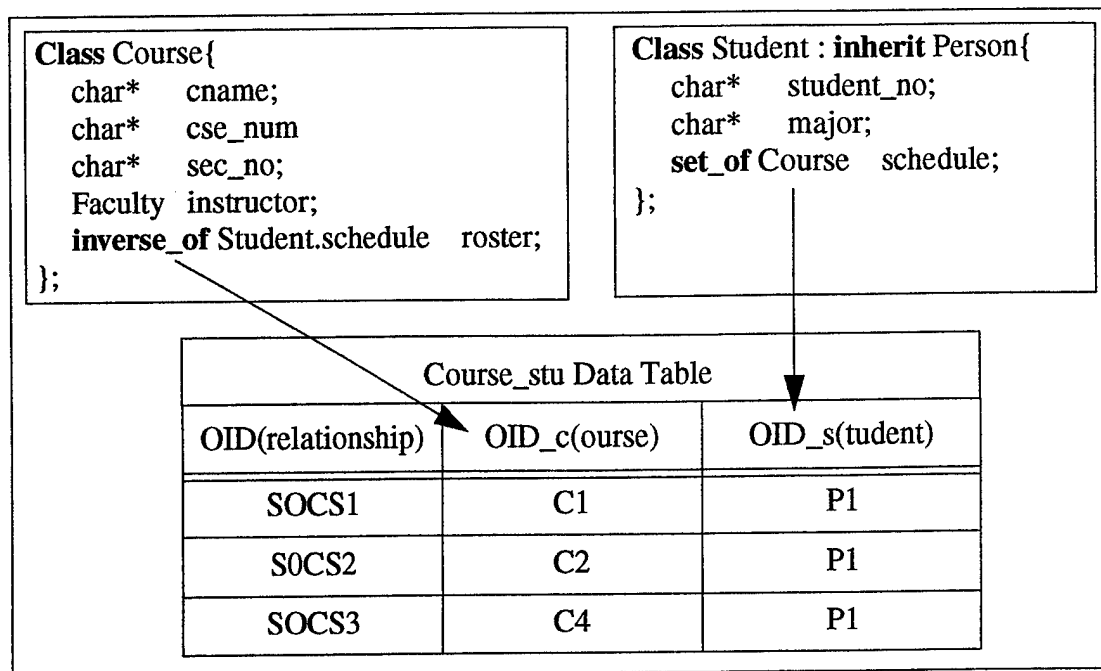
Figure 10. Set-of/Inverse-of Attribute-Set Relationships

additions or deletions to the Course_stu table will automatically update both objects' attribute-sets, thus maintaining data integrity and reducing redundancy.

2. The Delete Operation

The *Delete Operation* is an example of the concept of polymorphism and operator overloading adapted from the object-oriented paradigm. Here, the Delete Operation is overloaded. In addition to the destruction of objects first described in Chapter III.A.2, the Delete Operation is used here to remove a single_value attribute from an attribute-set. In this usage, delete does not imply destruction. The object removed from the attribute-set remains in the database and may be assigned to other objects and their attribute-sets. The format of the Delete Operation for an attribute-set is as follows:

delete (*set_attr, single_value*);

This operation utilizes the same underlying data structure and internal tables discussed in paragraph F.1 for the Insert Operation. For the delete operation, the set_attr

36

identifies the object OID and which attribute-set (and internal table) the relationship is recorded in. The single_value identifies which attribute in the attribute-set is to be deleted. With this information, the correct object in the internal table is located and deleted. Note that it is the relationship deleted, the object (identified by the single_value) is not deleted from the database. Note that this underlying data structure and operation is not visible to the user. To the user, his transaction is simply removing an attribute from an object's attribute-set.

3. The Contains Statement

The *Contains Statement* provides a unique search capability for an object-oriented database containing attribute-sets. By design the Contains Statement acts as a comparison operation, searching for those attributes in a set that match some specified criteria. For example, the Contains Statement will search an attribute-set for an attribute that matches a given expression and return any object or objects (OIDs) that satisfies the expression. The formal syntax for a Contains Statement is as follows:

> *set_attr* contains expression;
> or
> *set_attr* contains *obj_ref*;

Again, the underlying data structure and internal tables are used in conducting the search for matching OIDs and attributes satisfying a given expression.

4. Statistical Operations

Included in the design of O-ODML are certain *Statistical Operations* commonly used in database manipulations to enhance the ability of the database to perform grouping operations. These Statistical Operations are commonly referred as *aggregate functions.* Aggregate functions are used to specify mathematical manipulations on collections of values from the database. Common statistical operations include, *minimum, maximum,*

average and *count*. The count operation returns the number of values located in either the class or the attribute-set of an object. Note that Count will return an integer value, the total of all values located, counting duplicate values if found. The syntax and format for each query are as follows:

min(set_attr);

max(set_attr);

avg(set_attr);

count(set_attr); or count(class_name);

IV. MANIPULATION LANGUAGE SPECIFICATIONS RULES

A high-level programming language allows the programmer to write programs that resemble English or everyday mathematics. The user can use descriptive names that are easy to understand and describe operations using familiar symbols. Generally, a high-level language is easier to use; however, a machine language is the language of the computer and a high-level language must be translated before it can be executed. [Ref. 5]

A machine readable, high-level query language is designed with rules for syntax, grammar and production. These rules are required so that a machine, such as a compiler, may translate the symbolic language produced by the source query language into the proper commands of the target language [Ref. 7]. In the object-oriented data model/language (O-ODM), the manipulation language (O-ODML) is the source language and the kernel language of the M^2DBMS, the Attribute Based Database Management System (ABDBMS) is the target language.

In developing the language of the O-ODML, the specifications of the manipulation language include definitions, symbols, reserved words, grammar, syntax rules and production rules. These rules are well defined and specific. This permits a machine to read the high-level language of the user and convert the query into code that can be correctly translated into commands in the target language. If an error is made in applying the rules, a compilation error will result and the execution of the query will halt.

In addition, control structures such as conditional statements and loops are provided to enhance the abilities of the query programmer to manipulate the database, i.e., to operate on more than one record at a time.

A complete list of symbols, reserved words, production rules and definitions are provided as references in Appendixes A through D.

39

A. SPECIFICATIONS RULES

1. Symbols

Symbols are made up of the basic ASCII character set and are used by the query language to form commands and provide special meanings to the O-ODML compiler. Briefly, they include all the letters of the alphabet, all the digits and most of the special characters of the ASCII set. Not all the special characters have been adopted for use as symbols by the O-ODML, but are available for other purposes, such as forming an identifier or as part of a test string literal.

Each character is represented internally by its own unique ASCII numeric code. The printable characters range from 32 to 126. The table of ASCII numeric code is widely available and not included here; however, a few of its of its features include the following:

- Digits are in increasing value of consecutive characters and grouped together
- Uppercase letters are grouped in consecutive order and in increasing value
- Lowercase letters are grouped in consecutive order and in increasing value
- Digits precede uppercase letters that precede lowercase letters
- Special characters are interspersed in the basic ASCII set between 0 and 126

Each ASCII character of a query is read by scanning the input stream of characters and grouping them into identifiers and symbols. Identifiers and symbols are used by the compiler to construct the manipulations used by the target language to query the database. The implementation details of this process have been developed and are described in Master's Thesis by Carlos Barbosa, Aykut Kutlusan and Erhan Senocak. [Refs. 9, 11]

The O-ODML character set includes the following:

- Uppercase letters

    A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- Lowercase letters

a b c d e f g h i j k l m n o p q r s t u v w x y z

- Digits

    0 1 2 3 4 5 6 7 8 9

- Special characters

    ~ ' ! @ # $ % ^ & * ( ) _ - + = | \ { } [ ] " ' : ; < > , . ? /

- Non printable characters

    space, tab, end-of-line, end-of-file, carriage_return

Each of these characters may be taken together or individually to indicate a special symbol and meaning or to represent an identifier in the constructs of a query.

2. Identifiers

An *identifier* is used as a name for different components in a program, such as a procedure or a variable. Identifiers are also used to denote reserved words. The following rules govern the appearance of identifiers so they may be read by the compiler as such:

- Consist of a series of one or more characters
- Number of characters is not limited
- First character must be a letter
- Remaining characters may be letters, digits, underscore
- More than one underscore in succession is not allowed
- An underscore is not allowed as the final character
- Uppercase and lowercase are interchangeable and interpreted as the same thing, i.e., the O-ODML is not case sensitive
- First character following an embedded period in an identifier must be a letter

3. Reserved Words

Reserved words are identifiers reserved for special significance in the object-oriented data model/language and cannot be used for other purposes. A reserved word must not be used as a declared identifier. The list of reserved words is contained in Table 2.

ADD	END	IT
AND	END_IF	MAX
AVG	END_LOOP	MIN
BEGIN	FIND_MANY	MOD
CHAR	FIND_ONE	NOT
CHAR*	FLOAT	NULL
CLASS	FOR	OR
CONTAINS	IF	PROJECT
COUNT	IN	READ_INPUT
COVER	INHERIT	SET_OF
DELETE	INSERT	THEN
DISPLAY	INTEGER	QUERY
EACH	INVERSE_OF	WHERE
ELSE	IS	

Table 2. Reserved Words

- A reserved word must not be used as a declared identifier

- MOD is a Multiplication_Operator

- AND & OR are Logical_Operators

B. DECISION STATEMENTS

Decision statements are algorithms with two or more alternate paths of executions. The normal procedure is for a query to execute in sequential steps. There may be occasions when the programmer desires to provide alternate steps that may or may not be executed, depending on the results of previous transactions or based on input data. In the O-ODML, the decision on which path to execute is determined by evaluating a Boolean expression within a decision statement. A decision statement can be either a conditional statement or an iteration loop.

1. The If-Then Statement

The If-Then statement is a *conditional statement* and always contains a Boolean expression and at least one statement. The conditional statement determines if the sequence

of a statement or statements bounded by the If-Then statement is executed. Those statements bounded by the If-Then statement are statements included between the first reserved word If-Then and completion of the conditional statement signified by the last reserved word ENDIF. If the Boolean expression is evaluated as true, the statements contained within the If-Then statement are executed. If the Boolean expression is evaluated as false, the If-Then statement is bypassed and the next sequential statement following the conditional statement is then executed. [Ref. 6] The If-Then statement is written as follows:

IF (*Boolean expression*) THEN
    one or more statements;
ENDIF;

2. The If-Then-Else Statement

The If-Then-Else statement is closely related to the If-Then statement. The difference is that the If-Then-Else statement mandates that a choice of statements must be made and executed before continuing with the next sequential statement of the query following the conditional statement. [Ref. 6] The If-Then-Else statement is written as follows:

IF (*Boolean expression*) THEN
    first one or more statements;
ELSE
    second one or more statements;
ENDIF;

If the Boolean expression is evaluated as true, then the first sequence of statements are executed and the second sequence of statements are bypassed. If the Boolean expression is evaluated as false, then the first sequence of steps are bypassed and the second sequence of statements are executed. In the If-Then-Else statement, either the first or the second sequence of steps must be executed prior to continuing with the next statement or series of statements in the query. [Ref. 6]

C. ITERATIONS AND THE FOR-EACH LOOP STATEMENT

The *Iteration Statement*, the *For-Each Loop*, allows a statement or series of statements to be repeated a specified number of times. The number of repetitions is determined by some controlling condition. The For-Each Loop uses an *index* variable and a *control variable* as the condition to determine the number of times the statements within the loop will be repeated. [Ref. 6]

An index variable is an individual OID from the set of OIDs contained in the control variable. A control variable is the set of OIDs on which each iteration of the For-Each loop will act upon. An index variable must be declared as an object reference (obj_ref) type. A control variable must be declared as an object set (obj_set) type. For each iteration of the For-Each loop, the index variable will be assigned an individual value of an OID in the set of OIDs contained in control variable. Once the loop has been executed for each OID value of the control variable, the loop is terminated and control returns to the next sequence of statements in the query following the iteration statement. The following is an example of the For-Each Loop:

```
For Each index In control variable
    one or more statements;
End_Loop;
```

The index variable may be used in the body of the For-Each Loop. However, it cannot be modified beyond its defined iteration of the set_of control variable for each execution of the loop. [Ref. 5] *End_Loop* is a reserved word and signifies the completion of the For_Each Loop.

D. MEMBER ACCESS OPERATOR RULES

Commonly referred to as dot notation, an object's attributes may be referenced using a *member access selector*. This selector consists of an object's variable_name or

identifier, followed by the attribute_name, separated by a period. In the following example, Class Name may have an object stored in the database with an identifier of "thename" representing the OID of the object.

```
Class Name{
    char*    fname;
    char*    mi;
    char*    lname;
};
```

Each attribute may then be accessed as follows:

```
thename.fname
thename.mi
thename.lname
```

This method of referencing an attribute may also be used to assign new values to the attributes as well. For example:

```
thename.fname := 'Mike';
```

In the same manner, a method may be accessed to manipulate data.

thename.*method_name*(parameter_list);

where *method_name* would be a predefined method of the Class Name.

Now consider the case where an object is assigned as an attribute of another object.

```
Class Person{
    Name     pname;
    Address paddress;
    char*    sex;
};
```

In this example, Class Person has three attributes, pname, paddress and sex. Two of the attributes are of Class attribute_types, Name and Address, i.e., Name and Address are both a Class. Therefore, we have two Classes assigned as attribute-objects of another class. Note, this is different from the attribute-set previously defined, where objects are members of an attribute-set that is a class attribute. In this definition, the Class Name and Class

Address are attributes of the Class Person and are considered member objects. This relationship was previously defined as a complex object. In this situation, an object of Class Person will have an attribute pname of a data type of Class Name and an attribute of paddress of a data type of Class Address.

To access the attributes of an object of Class Name, of an object of Class Person, with an object reference variable name of "theperson", the dot notation would take the following form:

```
theperson.thename.fname
theperson.thename.mi
theperson.thename.lname
```

E. COMMENTED CODE

A query is a collection of programming code and as in any good program, it is important that provisions be made to allow the programmer to *comment* the code. Programmers insert comments to document programs and to improve program readability. They help other users to read and to understand the program. Comments do not cause the computer to perform any action, i.e., when the program is run, the comments are ignored by the compiler and do not cause any executable code to be generated. A comment that begins with "//" is called a single-line comment because the comment terminates at the end of the current line. [Ref. 6] An example of its use is as follows:

```
//This query will create a new object

Query InsertPerson IS
    obj_ref  p;          //declares p as a new variable of type obj_ref
Begin

    p := insert Person;  //assigns OID of new person to p
    p.sex := 'M';        //note that a literal only uses a single quote

End;
```

46

V. THE QUERY CONSTRUCTS

A *query* is a collection of reserved words, symbols and statements. Each query is structured in a block format to form declarations and database manipulations. Each forms a small program that must be properly coded to comply with the rules of syntax and grammar of the query language. To ensure the query is machine readable and able to compile, the structure and production rules of a query are well defined.

A. THE QUERY FORMAT

The format of a query is divided into five basic parts as illustrated in Table 3.

	Syntax	Semantics	Example
Part 1:	**QUERY id IS**	Query Heading	Query InsertPerson IS
Part 2:	*type* **id**;	Declarations Part	obj_ref p;
Part 3:	BEGIN	Reserved Word BEGIN	Begin
Part 4:	statement_list;	**Body** of *executable statements* and *operations*	p := insert Person; p.sex := 'M';
Part 5:	END;	Reserved Word END	End;

Table 3. The Constructs of a Query

The following example illustrates the format of a query:

```
Part 1:   Query InsertPerson IS
Part 2:        obj_ref    p;
Part 3:   Begin
Part 4:        p := insert Person;
               p.sex := 'M';
Part 5:   End;
```

47

1. Query Headings

The *Query Heading* consists of the reserved word QUERY, an identifier provided by the user, and the reserved word IS. The compiler recognizes the reserved word QUERY as the start of a legitimate program and proceeds to look for the identifier name to be assigned to the query, followed by the reserved word IS.

The identifier name for the query is generated by and for the benefit of a user to name or to describe the general purpose of the query. The query identifier must conform to the same syntactic rules as previously defined for an identifier in Chapter IV.A.2.

- Consist of a series of one or more characters
- Number of characters is not limited
- First character must be a letter
- Remaining characters may be letters, digits, underscore
- More than one underscore in succession is not allowed
- An underscore is not allowed as the final character
- Uppercase and lowercase are interchangeable, they are interpreted the same, i.e., the O-ODML is not case sensitive
- First character following an embedded period in an identifier must be a letter

Part 1 is completed when the reserved word IS, is compiled. A semicolon is not used to terminate the Query Heading.

2. Declarations Part

The *Declarations Part* is used to define variables and their data types. If there are no declarations, this part may be empty. Variable declarations provide the names of identifiers (also referred to as an *id*) that will be used to reference data items as defined by their type declarations or some predefined identifier. Variable names or identifiers must conform to the same syntactic rules described in paragraph 1. Data types or predefined identifiers include the following:

- obj_ref

- obj_set

- integer

- float

- char* (character strings)

One or more spaces are inserted between the data type and the identifier. If more than one identifier is declared of the same type, the identifiers may be listed consecutively on the same line, but must be separated by a comma. The last identifier on each line is followed by a semicolon, terminating that declaration. A new declaration may be made following the semicolon. A good programming practice is to have each new declaration begin on a new line. The following are examples of the formats that may be used in the declarations section:

```
obj_ref    first_id, second_id;    //commented code may appear here
obj_set    third_id, fourth_id;    //two variable may be declared
char*      fifth_id, sixth_id;     //semicolon follows each declaration
integer    seventh_id;            //each variable may be declared on a
integer    eigth_id;             //separate line if desired
float      ninth_id;
```

The Declarations Part concludes when the reserved word BEGIN is compiled. The reserved word BEGIN is used by the compiler to mark the completion of the Declarations Part and the beginning of the Body Part of the query. A semicolon is not used to terminate this line.

3. Body Part

The Body Part is composed of a series of executable statements. These statements include methods and operations as well as programming constructs of the manipulation language. The types of available operations were previously defined and are listed in Table 1, List of Object-Oriented Operations.

Statements can be further decomposed into simple statements such as expressions, or structured statements such as If-Then statements or Loop statements. Structured statements include the database operations previously defined Chapter III. All statement and expressions must be terminated by a semicolon.

The Body Part is completed when the reserved word END is compiled. The query is then compiled, executed, and the program terminated.

B. THE QUERY SYNTAX DEFINITION

There are several methods to formally define the syntax of the O-ODM manipulation language. These include the Backus-Naur Form (BNF) and syntax charts. To ensure accuracy and consistency, the production rules were chosen to define the syntax for the O-ODML, as they also define the formal mechanics of the syntax for the O-ODML compiler. The production rules are defined and included in Appendix B.

The production rules were generated using the LR(k) parsing techniques. The "L" is for the left-to-right method of scanning the input, the "R" for constructing a right-side derivation in reverse. The k is for the number of lookahead input symbols used in making a parsing decision. The productions of a *grammar* specify the manner in which the terminals and nonterminals may be combined to form strings. A grammar defines the constructs and hierarchical structure of a programming language. [Ref. 7]

The Context-Free Grammar is the convention chosen for the *productions* of the O-ODML production rules. Each production consists of a nonterminal, followed by the symbol "::=", followed by a string of nonterminals and terminals. [Ref. 7]

A *terminal* is a token, either a single character or several characters forming a string of tokens to represent a reserved word. Terminals cannot be decomposed into a smaller production rule.

A *nonterminal* represents a variable and is a sequence of tokens that can be further broken down into additional production rules. Several examples of nonterminals found in the O-ODML production rules include *expression, statement, list, declaration, part* and *term*.

The Context-Free Grammar consists of the following four parts:

- A set of tokens or terminal symbols

- A set of nonterminals

- A set of productions. Each production starts with a nonterminal on the left-side, followed by a symbol "::=", meaning "can have the form", and a series of terminals and nonterminals for the right-side

- A start symbol

A production is said to be *for* a nonterminal listed on the left-side. The following conventions will be used for a production in the Production Rules of the O-ODML. [Ref. 7]

- Terminals will be listed in boldface, such as digits, symbols and reserved words

- Reserved words will be listed in boldface caps

- Nonterminals will be listed in italics

- A vertical bar " | " will mean a choice of either-or must be made

- All right-side productions must be completed and in succeeding order, however each nonterminal must first be completed prior to proceeding to the next terminal/nonterminal of a production

- Tokens consist of zero or more characters

- A string of zero characters is the token " ε ", meaning epsilon or empty string

To illustrate the use of the production rules, Figure 11 contains a partial listing of the O-ODML production rules and Figure 12 illustrates their use. The full set of production rules for the object-oriented data manipulation language are provided in Appendix B.

51

start	::= **QUERY id IS** *declarative_part body_part*

Type and variable production rules

declarative_part	::= ε | *declaration declarative_part*
declaration	::= *variable_decl*
variable_decl	::= *type id_list* ;
id_list	::= **id** *id_lists*
id_lists	::= ε | , *id_list*
type	::= **obj_ref** | **obj_set** | **INTEGER** | **FLOAT** | **CHAR***

Procedure declaration production rules

body_part	::= **BEGIN** *statement_list* **END;**
.	.
.	.

Figure 11. Syntax Production Rules
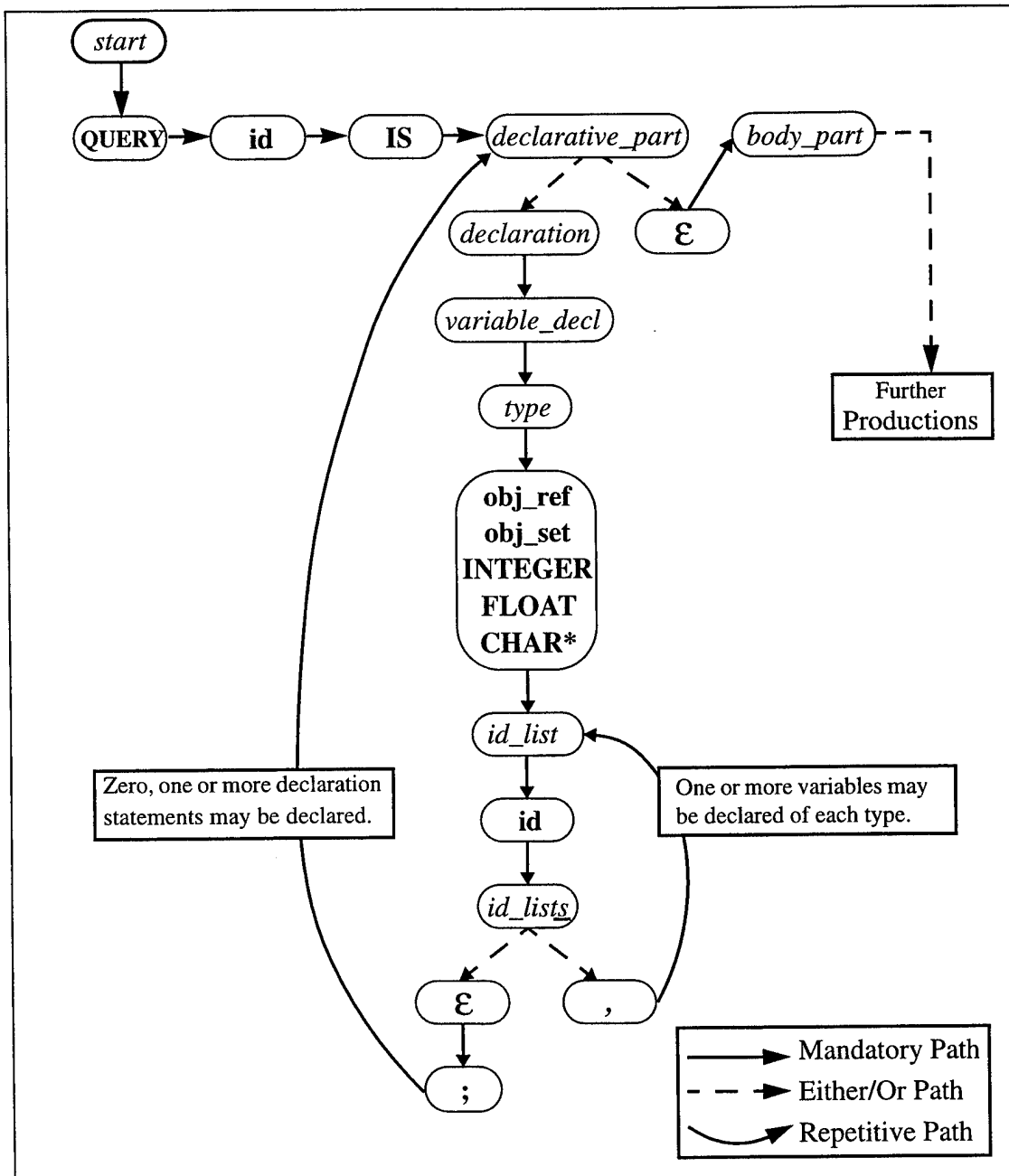
Figure 12. Production Rules Illustrated

VI. CONCLUSION

This thesis developed the design and specifications for an Object-Oriented Data Manipulation Language for an Object-Oriented Data Model. The Object-Oriented Data Model is one of five data models implemented as part of the Multimodel and Multilingual Database System database research project at Naval Postgraduate School at Monterey California. The object-oriented data manipulation language provides a high-level query language for writing and processing transactions that can be executed in the object-oriented language/paradigm via the Object-Oriented Model/Language interface.

The Object-Oriented Data Model is particularly well suited for modern database applications such as images and graphics databases, scientific databases, engineering design and manufacturing (CAD/CAM and CIM). These types of applications require complex objects capable of storing images or large textual items and defining nonstandard application-specific operations.

The Object-Oriented Data Model is based on the object-oriented programming language paradigm, and many of the constructs and features of the O-ODM were adopted from the programming language. These features include inheritance, encapsulation, polymorphism, operator overloading, composition and abstraction. Adopted for a database model, these features offer greater flexibility and efficiency to the programmer in creating a data base capable of emulating real world and evolutionary concepts.

The Object-Oriented Data Manipulation Language developed specifications to create and delete database objects, perform search and retrieval operations and input/output functions. Specifications were designed to manipulate the object-oriented relationships of inheritance, composition and covering. Programming language features borrowed from the object-oriented language include decision statements, member access functions and loop iterations.

The Object-Oriented Data Model has been successfully completed and satisfactorily demonstrated.

A. RELATED WORK

For a complete description and implementation details of the Object-Oriented Data Definition Language (O-ODDL), the reader is referred to the following Master's Thesis:

- The Design and Specification of an Object-Oriented Data Definition Language (O-ODDL) by Bruce Badgett [Ref. 4]

- The Design and Implementation of a Compiler for the Object-Oriented Data Definition Language (O-ODDL Compiler) by Luis Ramirez and Recep M. Tan [Ref. 12]

- The Instrumentation of a Kernel DBMS for the Support of a Database in the O-ODDL Specification by Dan Kellett and T. Kwon [Ref. 13]

For a complete description of the implementation details of the Object-Oriented Data Manipulation Language (O-ODML), the reader is referred to the following Master's Thesis:

- The Design and Implementation of a Compiler for the Object-Oriented Data Manipulation Language (O-ODML Compiler) by Carlos Barbosa and Aykut Kutlusan [Ref. 9]

- The Instrumentation of a Kernel DBMS for the Execution of Kernel Transactions Equivalent to their O-O Transactions by Robert Clark and Necmi Yildirum [Ref. 10]

- The Design and Implementation of a Real-Time Monitor for the Execution of Compiled Object-Oriented Transactions (O-ODDL and O-ODML Monitor) by Erhan Senocak [Ref. 11]

The reader is referred to following references for additional information on the Multimodel and Multilingual Database Management System.

- "The Object Oriented Database Management: A Tutorial On Its Fundamentals" by Dr. David K. Hsiao [Ref. 1]

- "A Parallel, Scalable, Microprocessor-Based Database Computer for Performance Gains and Capacity Growth" by Dr. David K. Hsiao [Ref. 14]

- The Multimodel and Multilingual Database System User's Manual by Paul Alan Bourgeois [Ref. 15]

B. RECOMMENDATIONS FOR FUTURE WORK

While many of the features of an object-oriented language were available, only a subset of those features were developed for the data model. While loops, goto statements, pointer functions, arrays, exception handling capabilities and other programming language features were not required for this implementation, but the potential for enhancing the capabilities of the O-ODML are easily seen. However, any additional features developed in the language specifications will also require additional coding of the O-ODML compiler and the Real-Time Monitor to ensure the correct mapping to the kernel database system.

Initial design decisions precluded the implementation of encapsulated methods. This is an important feature of the object-oriented paradigm and future work on the O-ODM should consider implementation of methods to provide enhanced external interface capability.

APPENDIX A. SYMBOLS

Addition_Operator	+ I -
Assign_Operator	:=
Close_Parenthesis	)
Colon	:
Comma	,
Comment	// till end of line
Delimiter	(SPACE I TAB I EOL) +
End_Of_File	
Epsilon	ε
Float_Constant	digit+ (digit+)* . digit+(digit+)* (No embedded underscores).
Identifier	letter+((_ (letter I digit))I(letterIdigit))* (. letter+(((_ (letter I digit))I(letterIdigit)))*)*
Integer_Constant	digit+((digit+) I (digit+))* (No embedded underscores).
Logical_Operator	AND I OR
Multiplication_Operator	* I / I MOD
NULL	
Open_Parenthesis	(
Relation_Operator	= I /= I < I <= I >= I >
Reserved_Word	See Table 2.
SemiColon	;
String_Constant	"printable chars, ASCII 32-126, and TAB"
Unary_Operator	-

Table 4. O-ODML Language Symbols

Key: *....................... Means 0 or more   I .... Separates options
        +....................... Means 1 or more   digit................. 0..9
        ().................... Groups of options, select one. letter.............. Means A-Z or a-z

59

APPENDIX B. PRODUCTION RULES

start   ::= **QUERY id IS** *declarative_part body_part*

Type and variable production rules

declarative_part   ::= ε | *declaration declarative_part*
declaration   ::= *variable_decl*
variable_decl   ::= *type id_list*;
id_list   ::= **id** *id_lists*
id_lists   ::= ε |, *id_list*
type   ::= **obj_ref | obj_set | INTEGER | FLOAT | CHAR***

Procedure declaration production rules

body_part   ::= **BEGIN** *statement_list* **END;**
statement_list   ::= *statement statement_lists*
statement_lists   ::= ε | *statement_list*
statement   ::= *simple_statement*; | *structured_statement*;

Simple statement production rules

simple_statement   ::= **id** *simple_statements* | **NULL**
simple_statements   ::= *actual_parameters* | **:=** *assign_statement*
actual_parameters   ::= ε | (*actual_parameter_list*)
actual_parameter_list   ::= *expression actual_parameter_lists*
actual_parameter_lists   ::= ε |, *actual_parameter_list*

Structured statement production rules

structured_statement   ::= *if_statement* | *loop_statement* | *delete_statement* |
                                  input_statement | *output_statement* | *add_statement*
assign_statement   ::= *insert_statement* | *find_statement* | *add_statement* | *expression*
if_statement   ::= **IF** *expression* **THEN** *statement_list else_part* **ENDIF**
else_part   ::= ε | **ELSE** *statement_list*
loop_statement   ::= **FOR EACH id IN id** *statement_list* **END_LOOP**
insert_statement   ::= **INSERT id**
delete_statement   ::= **DELETE** (*delete_parameter_list*)
delete_parameter_list   ::= **id | id, id**
find_statement   ::= **FIND_ONE id WHERE** *expression* |
                                  FIND_MANY id *where_expr*
where_expr   ::= ε | **WHERE** *expression*
input_statement   ::= **READ_INPUT** (*id_list*)
output_statement   ::= **PROJECT id | DISPLAY** (*display_parameter_list*)
display_parameter_list   ::= **id** | *literal*
add_statement   ::= **ADD (id, id)**
statistical_statement   ::= **MIN (id) | MAX (id) | AVG (id) | COUNT (id)**

Expression production rules

expression	::= *rel_expr expressions*
expressions	::= ε I **logical_operator** *expression* I *contain_expr*
rel_expr	::= *simple_expr rel_exprs* I *statistical_statement rel_exprs*
rel_exprs	::= ε I **relational_operator** *simple_expr*
contain_expr	::= **CONTAINS** *contain_exprs*
contain_exprs	::= *rel_expr*
simple_expr	::= *term simple_exprs*
simple_exprs	::= ε I **addition_operator** *simple_expr*
term	::= *factor terms*
terms	::= ε I **multiplication_operator** *term*
factor	::= **NOT** *primary* I *primary*
primary	::= *literal* I **id** I (*expression*)
literal	::= **integer_constant** I **float_constant** I **string_constant**

Note 1: Key to production rules: (1) *Nonterminals are in italics*, (2) RESERVED WORDS ARE IN BOLD UPPERCASE, and (3) token types are in bold lowercase.

Note 2: The Data Manipulation Language is not case sensitive.

APPENDIX C. DATA MANIPULATION LANGUAGE DEFINITIONS

Name	Semantics
attribute	A property that further describes the object, defined by the type of data it represents (attribute_type) and its identifier (attribute_name)
attribute_name	An identifier or name for the attribute
attribute_type	Predefined data type, may be either of a primitive or complex data type
attribute-object	An object, contained as an attribute in another object forming a complex class
attribute-set	An attribute of an object, the attribute being a set of objects, either a *set_of* or *inverse_of* attribute
class_name	The proper name of an object type, i.e., Class Person, Class Student
control variable	The set of OIDs through which a loop will cycle while performing some action
identifier	A variable name of either a reserved word, a variable or a query
index	Loop control variable, initialized to 1st OID of *control variable* when the FOR statement is executed and incremented through each OID in the control variable for each execution of the loop.
inverse_of	A predefined data type representing zero or more OIDs in an attribute-set, a set notation, the compliment of set_of
method_name	An object member, the identifier used to identify the method within the object formerly defined by a class
nonterminal	Variables that represent a sequences of tokens and defines <u>sets of strings</u> that help define the language generated by the grammar, *i.e., statement, expression.* In a parse tree, nonterminals are denoted as leafs.
obj_ref	A predefined data type that represents a single Object Identifier (OID) of a class, used specifically as the data type in the Find_One queries.
obj_set	A predefined data type that represents a set of Object Identifiers (OID), of multiple class objects used specifically as the type in Find_Many queries.
return_type	The type of data an object method will return upon completion of its transaction

Table 5. Definitions

Name	Semantics
set_attr	The OID and attribute of an object which is of data type *set_of* or *inverse_of* used to represent which attribute-set will be modified in an add or delete set operation.
set_of	A predefined data type representing zero or more OIDs in an attribute-set, a set notation, the compliment of inverse_of
single_value	The single object in a *set_attr* of objects.
string_constant	A string of characters used to display a message to the user. The string must be enclosed in double quotation marks.
string_literal	A string of characters used when data of type char* is assigned to an attribute. Must be enclosed in single quotation marks.
terminals	The basic symbols from which strings are formed. *i.e. if, then else.* In a parse tree, terminals are denoted as interior nodes.
var_list	A list of one to many variables that can receive data from the user through Read_Input Operation which can then assign the data to appropriate attributes in objects.
variable_name	An identifier, predefined as an object reference to represent one OID

Table 5. Definitions

APPENDIX D. LEXICAL NOTES AND SEMANTIC NOTES

Lexical Notes

1. String may have 0 to 250 characters, to place a " (quote) in a string, two quotes are needed.

2. **MOD** is a Multiplication_Operator, **AND** & **OR** are Logical_Operators.

Semantic Notes

1. All variables must be declared before they are referenced.

2. Duplicate identifiers may not be declared in the same scope.

3. The *expression* of an **IF** statement must be of type **BOOLEAN**.

4. **AND** and **OR** operators must be of type **BOOLEAN**.

5. +, -,*, / operators may be of type integer or float but must match.

6. Relational Operators must have integer operands.

7. The MOD operator must have integer operands.

8. Recursion is allowed.

9. Type checking is performed in assignment statements (Left Hand Side and Right Hand Side must be of the same type.).

APPENDIX E. FACULTY-STUDENT DATABASE LOGICAL DIAGRAM
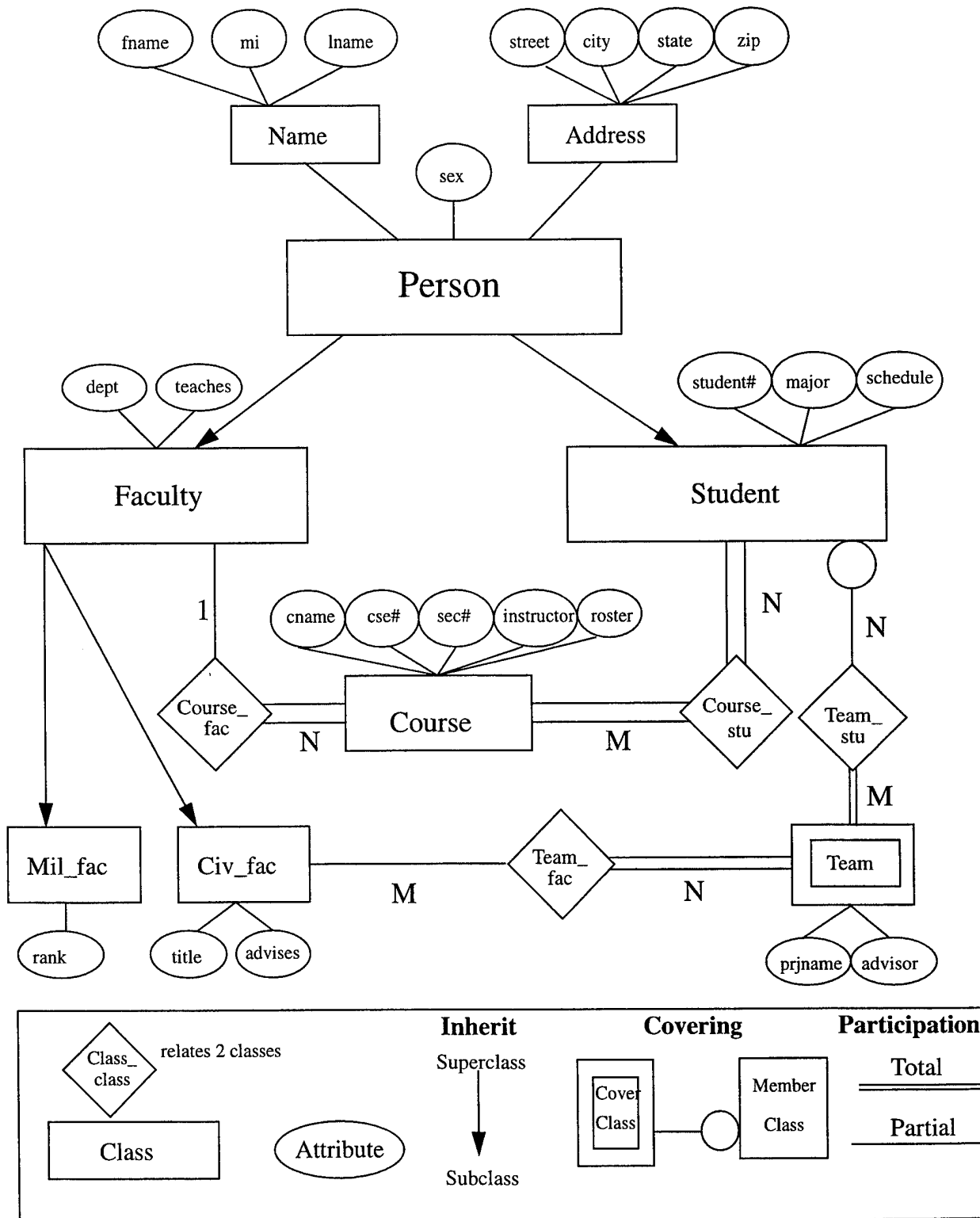
APPENDIX F. FACULTY-STUDENT DATABASE CLASS SPECIFICATION AND REPRESENTATION

Class Name{
    char*   fname;
    char*   mi;
    char*   lname;
};

Name	OID	fname	mi	lname

Class Address{
    char*   street;
    char*   city;
    char*   state;
    char   zipcode;
};

Address	OID	street	city	state	zipcode

Class Person{
    Name   pname;
    Address paddress;
    char*   sex;
};

Person	OID	pname	paddress	sex

Class Student : **inherit** Person{
    char*   student_no;
    char*   major;
    set_of   Course   schedule;
};

Student	OID	student_no	major	schedule

Class Faculty : **inherit** Person{
    char*   dept;
    set_of   Course   teaches;
};

Faculty	OID	dept	teaches

Class Civ_fac : **inherit** Faculty{
    char*   title;
    inverse_of Team.advisor   advises;
};

Civ_fac	OID	title	advises

Class Mil_fac : **inherit** Faculty{
    char*   rank;
};

Mil_fac	OID	rank

Class Team : **Cover** Student{
    char*   prjname;
    set_of   Civ_Fac   advisor;
};

Team	OID	prjname	advisor

Class Course{
    char*   cname;
    char*   cse_num;
    char*   sec_no;
    Faculty   instructor;
    inverse_of   Student.schedule roster;
};

Course	OID	cname	cse_num	sec_no	instructor	roster

APPENDIX G. FACULTY-STUDENT DATABASE QUERY FILES

The following queries illustrate the use of the O-ODML in constructing database manipulations. These queries were successfully used to test and demonstrate the object-oriented data manipulation language on the Faculty-Student Database depicted in Appendixes E and F. Each query is currently saved in an individual ASCII file. The files are located in the "/greg" directory of the "MDBS" account at the M^2DBMS database research laboratory of NPS Monterey. Each query file name is given in the title bar at the start of each query.

```
//******* FACSTUoolreq1 ******** //
// Retrieval


Query Display_Course IS

    obj_set    a;
    obj_ref    i;

Begin

    a := find_many Course where
        instructor.pname.lname = 'Wu';

    For Each i IN a
        display(i.cname, i.cse_no);
    End_Loop;

End;
```

```
// ******* FACSTUoolreq2 ******* //
// Multiple retrieval


Query Display_Courses IS
    obj_ref    a, i;
    obj_set    b;

Begin

    a := find_one Student where
        pname.lname = 'Badgett'
        and major = 'CS';

    b := find_many Course where
        roster contains a
        and sec_no = '1';

    For Each i In b
        display(i.cname);
    End_Loop;

End;
```

```
// ****** FACSTUoolreq3 ******* //
// Covering Retrieval

Query Display_Team_Info IS
    obj_ref    i;
    obj_set    a, b;
Begin
    a := find_many Student where
        pname.fname = 'Luis'
        or pname.fname = 'Recep';

    b := find_many Team where it
        contains a; // Covering relationship

    For Each i IN b
        display(i.prjname);
    End_Loop;

End;
```

```
// ******* FACSTUoolreq4 ********//
// Single Retrieval & Update

Query Update_MilFac_Rank IS
    obj_ref    p;
Begin
    p := find_one Mil_fac where
        sex = 'M'
        and rank = 'LCDR';

    display(p.rank, p.pname.fname,
        p.pname.lname);

    p.rank := 'CMDR';

    display(p.rank);

End;
```

```
/******** FACSTUoolreq5 ******* //
// Set Comparison

Query Display_Civ_Fac IS
    obj_ref    a, i;
    obj_set    b;
Begin
    a := find_one Team
        where prjname = 'OOP';

    b := find_many Civ_fac
        where advises contains a;

    For Each i IN b
        Display(i.title, i.pname.lname);
    End_Loop;

End;
```

```
// ******* FACSTUoolreq6 ******* //
// Multiple Update

Query Update_Student_Info IS
    obj_set    a;
    obj_ref    i;
Begin

    a := find_many Student where
        major = 'CS';

    For Each i IN a
        i.major := 'EC';
        display(i.pname.fname,
            i.pname.lname, i.major);
    End_Loop;

End;
```

```
// ****** FACSTUoolreq7 ******* //
// Multiple retrieval

Query Display_faculty IS
    obj_set    p;
    obj_ref    i;
Begin
    p := find_many Faculty
        where dept = CS;

    For Each i IN p
        display(i.pname.fname,
                i.pname.lname);
    End_Loop;

End;
```

```
// ******* FACSTUoolreq8 *******//
// Single Retrieval & Update

Query Update_CivFac_Address IS
    obj_ref    a;
Begin
    a := find_one Civ_Fac where
        pname.lname = 'Hsiao'
        and title = 'Prof';

    a.paddress.street := '150_Leahy_St';
    a.paddress.city := 'Carmel';
    a.paddress.zipcode := '93943';

    display(a.paddress.street,
            a.paddress.city,
            a.paddress.zipcode);

End;
```

APPENDIX H. FACULTY-STUDENT DATABASE QUERY EXAMPLES

The following queries were drafted to test various design and specification issues in the design of the O-ODML. These examples are included here to further illustrate various methods of constructing database manipulations utilizing the O-ODML.

1. ADD STUDENT
```
Query AddStudent IS
    obj_ref     s;
    char*       x, y;
Begin
        s := insert Student;
        display ("Input the student number and major");
        read_Input (x, y);
        s.student# := x;
        s.major := y;
End;
```

2. DROP STUDENT I
```
Query DropStudent IS
    obj_ref     p;
Begin
        p := find_One Student
            where major = 'cs'
            and student# = '20';
        delete (p);
End;
```

3. DROP STUDENT II

```
Query DropStudent IS
    obj_set    c;
    obj_ref    d, i;


Begin

        c := find_Many Student
            where pname.lname = 'Smith'
            and major = 'cs';

        For Each i IN c
            display (i.student#, i.pname.fname);
        End Loop;

    //Once correct student # is located from the first query
        d := find_One Student
            where student# = '20';

        delete (d);

End;
```

4. CHANGE ADDRESS I

```
Query ChangeAddress IS
    obj_ref    a;
    char*      x, y, z;
Begin

        a := find_One Student
            where pname.lname = 'Smith'
            and pname.fname = 'John'
            and student# = '20';

        display ("Input the new street address, city and zip");
        read_Input (x,y,z);
        a.paddress.street := x;
        a.paddress.city := y;
        a.paddress.zip := z;
End;
```

5. CHANGE ADDRESS II   //Example for changing all attributes of an object class
```
Query ChangeAddress IS
    obj_set    s;
    obj_ref    i;
    char*      x, y, z, w;
Begin

        s := find_Many Students
            where paddress.city = 'Monterey';

        For Each i IN s
            display (i.pname.fname, i.pname.lname);
            display ("Input the new street, city, state and zipcode.");
            read_Input (x, y, z, w);
            i.paddress.street := x;
            i.paddress.city := y;
            i.paddress.zipcode := w;
        End_Loop;

End;
```

6. CHANGE GENDER   //change all females to males
```
Query ChangeGender IS
    obj_set    p;
    obj_ref    i;
    char*      x;
Begin

        p := find_Many Student
            where sex = 'F';
        display ("Enter the correct gender, M/F");
        read_Input (x);

        For Each i In p
            i.sex := x;                //student inherits from person, will display sex
        End_Loop;

End;
```

7. CHANGE COURSE I

```
Query ChangeAddress IS
      obj_set    c;
      obj_ref    i;
      char*      x;
Begin

          c := find_Many Course
              where sec# = '2'
              and cse# = '4114';

          For Each i IN c
              display (i.cname, i.cse#);
              display ("Enter the correct course name");
              read_Input (x);
              i.cname := x;
          End_Loop;

End;
```

8. CHANGE COURSE II

```
Query ChangeCourse IS
      obj_ref    a;
      char*      x;
Begin

          a := find_One Course
              where cse# = '20';

          display ("Input the new course number.");
          read_Input (x);
          a.cse# := x;

End;
```

9. CHANGE THE TEAM OF A STUDENT

```
Query ChangeTeam IS
    obj_ref    t, s, k;
    char*      x;
Begin
        t := find_One Team
            where prjname = 'DB5';
        s := find_One Student
            where pname.lname = 'Smith'
            and pname.fname = 'John';
        delete (t, s);                    //delete s (student) from t (team)
        k := find_One Team
            where prjname = 'DB12';
        add(k, s);                        //add s (student) to t (team)
End;
```

The following queries demonstrate the use of "set-of" methods on attribute-sets.

10. ADD TEAMS THAT A CIV_FAC ADVISES   //teams already exist

```
Query CivFacTeam IS
    obj_set    a;
    obj_ref    b;
Begin
        a := find_Many Team
            where prjname = 'DB5';
        b := find_One Civ_Fac
            where pname.lname = 'Wu';
        For Each i IN a
            add(b.advises, i);            //union of set a and set b
        End_Loop;
End;
```

11. DELETE TEAMS THAT CIV_FAC ADVISES

```
Query CivFacTeam IS
    obj_ref    a, b;
Begin
        a := find_One Civ_Fac
            where pname.lname = 'Wu'
            and pname.fname = 'Thomas';
        b := find_One Team
            where prjname = 'OOP'
            and Team contains advisor = a;
        delete (a.advises, b);
End;
```

12. ADD AN ADVISOR TO A TEAM

```
Query TeamAdvisor IS
    obj_ref     cy, cx;
Begin
        cy := find_One Team
            where prjname = 'DB5';
        cx := find_One Civ_Fac
            where pname.lname = 'smith'
            and pname.fname = "john";
        add (cy.advisor, cx);
End;
```

13. DELETE A SINGLE ADVISOR FROM THE SET OF ADVISORS OF A TEAM

```
Query CivFacTeam IS
    obj_ref     a, b;
Begin
        a := find_One Team
            where prjname = 'DB5';
        b := find_One Civ_Fac
            where pname.lname = 'Hsiao'
            and pname.fname = 'David';
        delete (a.advisor, b);
End;
```

14. ADD AN INSTRUCTOR TO A COURSE

```
Query InstructorCourse IS
    obj_ref     a, b;
Begin

    a := find_One Faculty
        where pname.lname = 'Wu';
        and dept = 'CS';
    b := find_One Course
        where cse# = '3320';
    add (b.instrucotr, a);
End;
```

15. DELETE AN INSTRUCTOR FROM A COURSE

```
Query CourseInstructor IS
    obj_ref      a, b;
Begin
        a := find_One Faculty
            where pname.lname = 'Wu';
            and dept = 'CS';
        b := find_One Course
            where cse# = '3320';
        delete (b.instrucotr, a);
End;
```


16. ADD A LIST OF COURSES TO A STUDENT'S SCHEDULE

```
Query StudentSchedule IS
    obj_ref      i, p;
    obj_set      c;
Begin
        p := find_One Student
            where pname.lname = 'Badgett'
            and student# = '20';
        c := find_Many Course
            where cse# = '4322' and sec# = '1'
            or cse# = '4114' and sec# = '1';

        For Each i IN c
            add (p.schedule, i);        //set union of courses to student's schedule
        End_Loop;
End;
```


17. DELETE A LIST OF COURSES FROM A STUDENT'S SCHEDULE

```
Query StudentSchedule IS
    obj_ref      i, p;
    obj_set      c;
Begin
        p := find_One Student
            where pname.lname = 'Badgett'
            and student# = '20';
        c := find_Many Course
            where cse# = '4322' and sec# = '1'
            or cse# = '4114' and sec# = '1';
        For Each i IN c
            delete (p.schedule, i);     //set difference of student's schedule minus courses
        End_Loop;
End;
```

18. DELETE LIST OF COURSES FROM WHAT A FACULTY MEMBER TEACHES

```
Query FacultyTeaches IS
    obj_ref    a, i;
    obj_set    b;
Begin

        a := find_One Faculty
            where pname.lname ='Wu'
            and pname.lname = 'Thomas';
        b := find_Many Courses
            where cname = 'OOPROG';

        For Each i IN a
            delete (i.teaches, b);
        End_Loop;

End;
```

LIST OF REFERENCES

[1]   Hsiao, David K., "The Object Oriented Database Management: A Tutorial On It's Fundamentals", *Proceedings of the Second Far-East Workshop on Future Database Systems*, Kyoto, Japan, April 1992.

[2]   Hsiao, David K., Wu, C. Thomas, "Interoperable and Multidatabase Solutions for Heterogeneous Databases and Transactions", Draft Notes in Computer Science, NPS Monterey, CA, July 1994.

[3]   Pohl, I., *Object Oriented Programming Using C++*, Benjamin/Cummings Publishing Company, Inc, 1993.

[4]   Badgett, R.B., *The Design and Specification of an Object-Oriented Data Definition Language (O-ODDL)*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.

[5]   Feldman, Michael B, Koffman, Elliot B., *Ada: Problem Solving and Program Design*, Addison-Wesley Publishing Company, 1993.

[6]   Deitel, Harvey M., Deitel, P.J., *C++ How To Program*, Prentice Hall, Inc., 1994.

[7]   Aho, Alfred V., Sethi, Ravi, Ullman, Jeffrey D., *Compilers, Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1988.

[8]   Elmasri and Navathe, *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, Inc., 1990.

[9]   Barbosa, C. and Kutlusan, A., *The Design and Implementation of a Compiler for the Object-Oriented Data Manipulation Language (O-ODML Compiler)*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.

[10]   Clark, R. and Yildirum, N., *The Instrumentation of a Kernel DBMS for the Execution of Kernel Transactions Equivalent to their O-O Transactions*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.

[11]   Senocak, E., *The Design and Implementation of a Real-Time Monitor for the Execution of Compiled Object-Oriented Transactions (O-ODDL and O-ODML Monitor)*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.

[12]   Ramirez, L. and Tan, R., M., *The Design and Implementation of a Compiler for the Object-Oriented Data Definition Language (O-ODDL Compiler)*, Master's Thesis,

Naval Postgraduate School, Monterey, California, September 1995.

[13]   Kellett, D. and Kwon, T., *The Instrumentation of a Kernel DBMS for the support of a Database in the O-ODDL Specification*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.

[14]   Hsiao, David K., "A Parallel, Scalable, Microprocessor-Based Database Computer for Performance Gains and Capacity Growth", IEEE Micro, December 1991.

[15]   Bourgeois, Paul Alan, *The Multimodel and Multilingual Database System User's Manual*, Masters Thesis, Naval Postgraduate School, Monterey, California, December 1992.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA   22304-6145

2. Dudley Knox Library 2
   Code 013
   Naval Postgraduate School
   Monterey, CA   93943-5101

3. Chairman, Code CS 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA   93943

4. Dr. David K. Hsiao, Code CS/HS 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA   93943

5. Dr. C. Thomas Wu, Code CS/KA 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA   93943

6. MS. Doris Mlezko 2
   Code P22305
   Weapons Division
   Naval Air Warfare Center
   Pt Mugu, CA 93042-5001

7. Ms. Sharon Cain 2
   NAIC/SCDD
   4115 Hebble Creek Rd
   Wright Patterson AFB, OH 45433-5622

8. Mr. D. W. Stephens 1
   4258 Drake St
   Houston, TX   77005

9. LCDR Michael W. Stephens 2
    4258 Drake St
    Houston, TX   77005

10. Mrs. Shirley Graham 1
    4921 S. E. 85th St
    Portland, OR   97266